Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt

Everton da Silva Maldonado, Emad Shihab, *Member, IEEE*, and Nikolaos Tsantalis, *Member, IEEE*

Abstract—The metaphor of technical debt was introduced to express the trade off between productivity and quality, i.e., when developers take shortcuts or perform quick hacks. More recently, our work has shown that it is possible to detect technical debt using source code comments (i.e., self-admitted technical debt), and that the most common types of self-admitted technical debt are design and requirement debt. However, all approaches thus far heavily depend on the manual classification of source code comments. In this paper, we present an approach to automatically identify design and requirement self-admitted technical debt using Natural Language Processing (NLP). We study 10 open source projects: Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby and SQuirrel SQL and find that 1) we are able to accurately identify self-admitted technical debt, significantly outperforming the current state-of-the-art based on fixed keywords and phrases; 2) words related to sloppy code or mediocre source code quality are the best indicators of design debt, whereas words related to the need to complete a partially implemented requirement in the future are the best indicators of requirement debt; and 3) we can achieve 90 percent of the best classification performance, using as little as 23 percent of the comments for both design and requirement self-admitted technical debt, and 80 percent of the best performance, using as little as 9 and 5 percent of the comments for design and requirement self-admitted technical debt, respectively. The last finding shows that the proposed approach can achieve a good accuracy even with a relatively small training dataset.

Index Terms—Technical debt, source code comments, natural language processing, empirical study

1 Introduction

Developers often have to deal with conflicting goals that require software to be delivered quickly, with high quality, and on budget. In practice, achieving all of these goals at the same time can be challenging, causing a tradeoff to be made. Often, these tradeoffs lead developers to take *shortcuts* or use *workarounds*. Although such shortcuts help developers in meeting their short-term goals, they may have a negative impact in the long-term.

Technical debt is a metaphor coined to express sub-optimal solutions that are taken in a software project in order to achieve some short-term goals [1]. Generally, these decisions allow the project to move faster in the short-term, but introduce an increased cost (i.e., debt) to maintain this software in the long run [2], [3]. Prior work has shown that technical debt is widespread in the software domain, is unavoidable, and can have a negative impact on the quality of the software [4].

- *E. da S. Maldonado and E. Shihab are with the Data-Driven Analysis of Software (DAS) Lab, Department of Computer Science and Software Engineering, Concordia University, Montreal, QC H4B 1R6, Canada. E-mail: {e_silvam, eshihab}@encs.concordia.ca.*
- *N. Tsantalis is with the Department of Computer Science and Software Engineering, Concordia University, Montreal, QC H4B 1R6, Canada. E-mail: tsantalis@encs.concordia.ca.*

Manuscript received 17 Dec. 2015; revised 2 Dec. 2016; accepted 6 Jan. 2017. Date of publication 16 Jan. 2017; date of current version 20 Nov. 2017. Recommended for acceptance by G.C. Murphy.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2654244

Technical debt can be deliberately or inadvertently incurred [5]. Inadvertent technical debt is technical debt that is taken on unknowingly. One example of inadvertent technical debt is architectural decay or architectural drift. To date, the majority of the technical debt work has focused on inadvertent technical debt [6]. On the other hand, deliberate technical debt, is debt that is incurred by the developer with knowledge that it is being taken on. One example of such deliberate technical debt, is self-admitted technical debt, which is the focus of our paper.

Due to the importance of technical debt, a number of studies empirically examined technical debt and proposed techniques to enable its detection and management. Some of the approaches analyze the source code to detect technical debt, whereas other approaches leverage various techniques and artifacts, e.g., documentation and architecture reviews, to detect documentation debt, test debt or architecture debt (i.e., unexpected deviance from the initial architecture) [7], [8].

The main findings of prior work are three-fold. First, there are different types of technical debt, e.g., defect debt, design debt, testing debt, and that among them design debt has the highest impact [9], [10]. Second, static source code analysis helps in detecting technical debt, (i.e., code smells) [11], [12], [13]. Third, more recently, our work has shown that it is possible to identify technical debt through source comments, referred to as self-admitted technical debt [14], and that design and requirement debt are the most common types of self-admitted technical debt [15].

The recovery of technical debt through source code comments has two main advantages over traditional approaches based on source code analysis. First, it is more lightweight compared to source code analysis, since it does not require the construction of Abstract Syntax Trees or other more advanced source code representations. For instance, some code smell detectors that also provide refactoring recommendations to resolve the detected code smells [16], [17] generate computationally expensive program representation structures, such as program dependence graphs [18], and method call graphs [19] in order to match structural code smell patterns and compute metrics. On the other hand, the source code comments can be easily and efficiently extracted from source code files using regular expressions. Second, it does not depend on arbitrary metric threshold values, which are required in all metric-based code smell detection approaches. Deriving appropriate threshold values is a challenging open problem that has attracted the attention and effort of several researchers [20], [21], [22]. As a matter of fact, the approaches based on source code analysis suffer from high false positive rates [23] (i.e., they flag a large number of source code elements as problematic, while they are not perceived as such by the developers), because they rely only on the structure of the source code to detect code smells without taking into account the developers' feedback, the project domain, and the context in which the code smells are detected.

However, relying solely on the developers' comments to recover technical debt is not adequate, because developers might be unaware of the presence of some code smells in their project, or might not be very familiar with good design and coding practices (i.e., inadvertent debt). As a result, the detection of technical debt through source code comments can be only used as a complementary approach to existing code smell detectors based on source code analysis. We believe that self-admitted technical debt can be useful to prioritize the *pay back* of debt (i.e., develop a *pay back* plan), since the technical debt expressed in the comments written by the developers themselves might be more relevant to them. As a matter of fact, in a recent survey [24] with 152 developers of a large financial organization (ING Netherlands), 88 percent of the participants responded that they annotate poor implementation choices (i.e., design technical debt) with comments in the source code (i.e., self-admitted technical debt), and when time allows, they act on them by trying to refactor such smells using some automated tool support (71 percent), or manually (29 percent).

Despite the advantages of recovering technical debt from source code comments, the research in self-admitted technical debt, thus far, heavily relies on the manual inspection of code comments. The current-state-of-the art approach [14] uses 62 comment patterns (i.e., words and phrases) derived after the manual examination of more than 100 K comments. The manual inspection of code comments is subject to reader bias, time consuming and, as any other manual task, susceptible to errors. These limitations in the identification of self-admitted technical debt comments makes the current state-of-the-art approach difficult to be applied in practice.

Therefore, in this paper we investigate the efficiency of using Natural Language Processing (NLP) techniques to automatically detect the two most common types of self-admitted technical debt, i.e., design and requirement debt. We analyze ten open source projects from different application domains, namely, Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby and SQuirrel SQL. We extract and classify the source comments of these projects. Then, using the classified dataset we train a maximum entropy classifier using the Stanford Classifier tool [25] to identify design and requirement self-admitted technical debt. The advantages of the maximum entropy classifier over keyword-based and pattern-based approaches, such as comment patterns, are twofold. First, the maximum entropy classifier automatically extracts the most important features (i.e., words) for each class (i.e., design self-admitted technical debt, requirement self-admitted technical debt, and without technical debt) based on a classified training dataset given as input. Second, the maximum entropy classifier, apart from finding features that contribute positively to the classification of a comment in a given class, also finds features that contribute negatively to the classification of a comment in a given class.

We perform a leave-one-out cross-project validation (i.e., we train on nine projects and test on one project). Our results show that we are able to achieve an average F1-measure of 0.620 when identifying design self-admitted technical debt, and an average F1-measure of 0.403 when identifying requirement self-admitted technical debt. We compare the performance of our approach to a simple (random) baseline and the state-of-the-art approach used to detect self-admitted technical debt [14]. Our results show that on average, we outperform the state-of-the-art by2.3 times, when detecting design debt, and by 6 times when detecting requirement debt.

To better understand how developers express technical debt we analyze the 10 most prevalent words appearing within self-admitted technical debt comments. We find that the top design debt words are related to sloppy or mediocre source code. For example, words such as 'hack', 'workaround' and 'yuck!' are used to express design self-admitted technical debt. On the other hand, for requirement debt, words indicating the need to complete a partially implemented requirement are the best indicators. For example, words such as 'todo', 'needed' and 'implementation' are strong indicators of requirement debt.

Finally, to determine the most efficient way to apply our approach, we analyze the amount of training data necessary to effectively identify self-admitted technical debt. We find that training datasets using 23 percent of the available data can achieve a performance equivalent to 90 percent of the maximum F1-measure score for both design and requirement self-admitted technical debt. Similarly, 80 percent of the maximum F1-measure can be achieved using only 9 percent of the available data for design self-admitted technical debt, and 5 percent for requirement self-admitted technical debt.

The main contributions of our work are the following:

- We provide an automatic, NLP-based, approach to identify design and requirement self-admitted technical debt.
- We examine and report the words that best indicate design and requirement self-admitted technical debt.
- We show that using a small training set of comments, we are able to effectively detect design and requirement self-admitted technical debt.

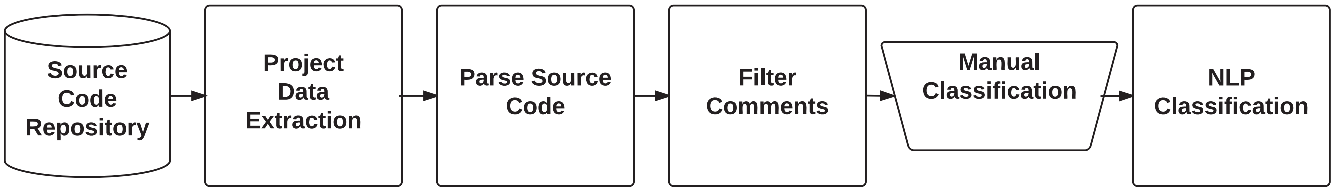

Fig. 1. Approach overview.

- We make our dataset publicly available, so that others can advance work in the area of self-admitted technical debt [26].

The rest of the paper is organized as follows. Section 2 describes our approach. We setup our experiment and present our results in Section 3. We discuss the implications of our findings in Section 4. In Section 5 we present the related work. Section 6 presents the threats to validity and Section 7 presents our conclusions and future work.

2 Approach

The main goal of our study is to automatically identify self-admitted technical debt through source code comments. To do that, we first extract the comments from ten open source projects. Second, we apply five filtering heuristics to remove comments that are irrelevant for the identification of self-admitted technical debt (e.g., license comments, commented source code and Javadoc comments). After that, we manually classify the remaining comments into the different types of self-admitted technical debt (i.e., design debt, requirement debt, defect debt, documentation debt and test debt). Lastly, we use these comments as training data for the maximum entropy classified and use the trained model to detect self-admitted technical debt from source code comments. Fig. 1 shows an overview of our approach, and the following sections detail each step.

2.1 Project Data Extraction

To perform our study, we need to analyze the source code comments of software projects. Therefore, we focused our study on ten open source projects: Ant is a build tool written in Java, ArgoUML is an UML modeling tool that includes support for all standard UML 1.4 diagrams, Columba is an email client that has a graphical interface with wizards and internationalization support, EMF is a modeling framework and code generation facility for building tools and other applications, Hibernate is a component providing Object Relational Mapping (ORM) support to applications and other components, JEdit is a text editor written in Java, JFreeChart is a chart library for the Java platform, JMeter is a Java application designed to load functional test behavior and measure performance, JRuby is a pure-Java implementation of the Ruby programming language and SQuirrel SQL is a graphical SQL client written in Java. We selected these projects since they belong to different application domains, are well commented, vary in size, and in the number of contributors.

Table 1 provides details about each of the projects used in our study. The columns of Table 1 present the release used, followed by the number of classes, the total source lines of code (SLOC), the number of contributors, the number of extracted comments, the number of comments analyzed after applying our filtering heuristics, and the number of comments that were classified as self-admitted technical debt together with the percentage of the total project comments that it represent. The final three columns show the percentage of self-admitted technical debt comments classified as design debt, requirement debt, and all other remaining types of debt (i.e., defect, documentation and test debt), respectively.

Since there are many different definitions for the SLOC metric we clarify that, in our study, a source line of code contains at least one valid character, which is not a blank space or a source code comment. In addition, we only use the Java files to calculate the SLOC, and to do so, we use the SLOCCount tool [27].

TABLE 1
Details of the Studied Projects

Project	Project Details				Comments Details			Technical Debt Details		
	Release	# of Classes	SLOC	# of Contributors	# of Comments	# of Comments After Filtering	#,(%) of TD Comments	% of Design Debt	% of Requirement Debt	% of Other Debt
Ant	1.7.0	1,475	115,881	74	21,587	4,137	131 (0.60)	72.51	09.92	17.55
ArgoUML	0.34	2,609	176,839	87	67,716	9,548	1,413 (2.08)	56.68	29.08	14.22
Columba	1.4	1,711	100,200	9	33,895	6,478	204 (0.60)	61.76	21.07	17.15
EMF	2.4.1	1,458	228,191	30	25,229	4,401	104 (0.41)	75.00	15.38	09.61
Hibernate	3.3.2 GA	1,356	173,467	226	11,630	2,968	472 (4.05)	75.21	13.55	11.22
JEdit	4.2	800	88,583	57	16,991	10,322	256 (1.50)	76.56	05.46	17.96
JFreeChart	1.0.19	1,065	132,296	19	23,474	4,423	209 (0.89)	88.03	07.17	04.78
JMeter	2.10	1,181	81,307	33	20,084	8,162	374 (1.86)	84.49	05.61	09.89
JRuby	1.4.0	1,486	150,060	328	11,149	4,897	622 (5.57)	55.14	17.68	27.17
SQuirrel	3.0.3	3,108	215,234	46	27,474	7,230	286 (1.04)	73.07	17.48	09.44
Average		1,625	146,206	91	25,923	6,257	407 (1.86)	71.84	14.24	13.89
Total		16,249	1,462,058	909	259,229	62,566	4,071 (-)	-	-	-

The number of contributors was extracted from OpenHub, an on-line community and public directory that offers analytics, search services and tools for open source software [28]. It is important to note that the number of comments shown for each project does not represent the number of commented lines, but rather the number of Single-line, Block and Javadoc comments. In total, we obtained 259,229 comments, found in 16,249 Java classes. The size of the selected projects varies between 81,307 and 228,191 SLOC, and the number of contributors of these projects ranges from 9 to 328.

2.2 Parse Source Code

After obtaining the source code of all projects, we extract the comments from the source code. We use JDeodorant [29], an open-source Eclipse plug-in, to parse the source code and extract the code comments. JDeodorant provides detailed information about the source code comments such as: their type (i.e., Block, Single-line, or Javadoc), their location (i.e., the lines where they start and end), and their context (i.e., the method/field/type declaration they belong to).

Due to these features, we adapted JDeodorant to extract the aforementioned information about source code comments and store it in a relational database to facilitate the processing of the data.

2.3 Filter Comments

Source code comments can be used for different purposes in a project, such as giving context, documenting, expressing thoughts, opinions and authorship, and in some cases, disabling source code from the program. Comments are used freely by developers and with limited formalities, if any at all. This informal environment allows developers to bring to light opinions, insights and even confessions (e.g., self-admitted technical debt).

As shown in prior work [15], part of these comments may discuss self-admitted technical debt, but not the majority of them. With that in mind, we develop and apply five filtering heuristics to narrow down the comments eliminating the ones that are less likely to be classified as self-admitted technical debt.

To do so, we developed a Java based tool that reads from the database the data obtained by parsing the source code. Next, it executes the filtering heuristics and stores the results back in the database. The retrieved data contains information like the line number that a class/comment starts/ends and the comment type, considering the Java syntax (i.e., Single-line, Block or Javadoc). With this information we process the filtering heuristics as described next.

License comments are not very likely to contain self-admitted technical debt, and are commonly added before the declaration of the class. We create a heuristic that removes comments that are placed before the class declaration. Since we know the line number that the class was declared we can easily check for comments that are placed before that line and remove them. In order to decrease the chances of removing a self-admitted technical debt comment while executing this filter we calibrated this heuristic to avoid removing comments that contain one of the predefined task annotations (i.e., “TODO:”, “FIXME:”, or “XXX:”) [30]. Task annotations are an extended functionality provided by most of the popular Java *IDEs* including Eclipse, InteliJ and NetBeans. When one of these words is used inside a comment the IDE will automatically keep track of the comment creating a centralized list of tasks that can be conveniently accessed later on.

Long comments that are created using multiple *Single-line* comments instead of a *Block* comment can hinder the understanding of the message considering the case that the reader (i.e., human or machine) analyzes each one of these comments independently. To solve this problem, we create a heuristic that searches for consecutive single-line comments and groups them as one comment.

Commented source code is found in the projects due to many different reasons. One of the possibilities is that the code is not currently being used. Other is that, the code is used for debugging purposes only. Based on our analysis, commented source code does not have self-admitted technical debt. Our heuristic removes commented source code using a simple regular expression that captures typical Java code structures.

Automatically generated comments by the IDE are filtered out as well. These comments are inserted as part of code snippets used to generate constructors, methods and try catch blocks, and have a fixed format (i.e., “Auto-generated constructor stub”, “Auto-generated method stub”, and “Auto-generated catch block”). Therefore our heuristic searches for these automatically generated comments and removes them.

Javadoc comments rarely mention self-admitted technical debt. For the Javadoc comments that do mention self-admitted technical debt, we notice that they usually contain one of the task annotations (i.e., “TODO:”, “FIXME:”, or “XXX:”). Therefore, our heuristic removes all comments of the Javadoc type, unless they contain at least one of the task annotations. To do so, we create a simple regular expression that searches for the task annotations before removing the comment.

The steps mentioned above significantly reduced the number of comments in our dataset and helped us focus on the most applicable and insightful comments. For example, in the Ant project, applying the above steps helped to reduce the number of comments from 21,587 to 4,137 resulting in a reduction of 80.83 percent in the number of comments to be manually analyzed. Using the filtering heuristics we were able to remove from 39.25 to 85.89 percent of all comments. Table 1 provides the number of comments kept after the filtering heuristics for each project.

2.4 Manual Classification

Our goal is to inspect each comment and label it with a suitable technical debt classification. Since there are many comments, we developed a Java based tool that shows one comment at a time and gives a list of possible classifications that can be manually assigned to the comment. The list of possible classifications is based on previous work by Alves et al. [9]. In their work, an ontology on technical debt terms was proposed, and they identified the following types of technical debt across the researched literature: architecture, build, code, defect, design, documentation, infrastructure, people, process, requirement, service, test automation and test debt. During the classification process, we notice that not all types of debt mentioned by Alves et al. [9] could be found in code comments. However, we were able to identify the following types of debt in the source comments: design debt, defect debt, documentation debt, requirement debt and test debt.

In our previous work [15], we manually classified 33,093 commentsextracted from the following projects: Ant, ArgoUML, Columba, JFreeChart and JMeter.R2 In the current study we manually classified an additional 29,473 commentsfrom EMF, Hibernate, JEdit, JRuby and SQuirrelR2, which means that we extended our dataset of classified comments by 89.06 percent. In total, we manually classified 62,566 comments into the five different types of self-admitted technical debt mentioned above. The classification process took approximately 185 hours in total, and was performed by the first author of the paper. It is important to note that this manual classification step does not need to be repeated in order to apply our approach, since our dataset is publicly available [26], and thus it can used as is, or even extended with new classified comments.

Below, we provide definitions for design and requirement self-admitted technical debt, and some indicative comments to help the reader understand the different types of self-admitted technical debt comments.

Self-Admitted Design Debt. These comments indicate that there is a problem with the design of the code. They can be comments about misplaced code, lack of abstraction, long methods, poor implementation, workarounds, or temporary solutions. Usually these kinds of issues are resolved through refactoring (i.e., restructuring of *existing* code), or by re-implementing *existing* code to make it faster, more secure, more stable and so forth. Let us consider the following comments:

> *"TODO: - This method is too complex, lets break it up"* - [from ArgoUml]
>
> *"/* TODO: really should be a separate class */"* - [from ArgoUml]

These comments are clear examples of what we consider as self-admitted *design debt*. In the above comments, the developers state what needs to be done in order to improve the current design of the code, and the payback of this kind of design debt can be achieved through refactoring. Although the above comments are easy to understand, during our study we came across more challenging comments that expressed design problems in an indirect way. For example:

> *"// I hate this so much even before I start writing it. // Re-initialising a global in a place where no-one will see it just // feels wrong. Oh well, here goes."* - [from ArgoUml]
>
> *"//quick & dirty, to make nested mapped p-sets work:"* - [from Apache Ant]

In the above example comments the authors are certain to be implementing code that does not represent the best solution. We assume that this kind of implementation will degrade the design of the code and should be avoided.

> *"// probably not the best choice, but it solves the problem of // relative paths in CLASSPATH"* - [from Apache Ant]
>
> *"//I can't get my head around this; is encoding treatment needed here?"* - [from Apache Ant]

The above comments expressed doubt and uncertainty when implementing the code and were considered as self-admitted design debt as well. The payback of the design debt expressed in the last four example comments can be achieved through the re-implementation of the currently existing solution.

Self-Admitted Requirement Debt. These comments convey the opinion of a developer supporting that the implementation of a requirement is not complete. In general, requirement debt comments express that there is still *missing* code that needs to be added in order to complete a *partially* implemented requirement, as it can be observed in the following comments:

> *"/TODO no methods yet for getClassname"* - [from Apache Ant]
>
> *"//TODO no method for newInstance using a reverse-classloader"* - [from Apache Ant]
>
> *"TODO: The copy function is not yet * completely implemented - so we will * have some exceptions here and there.*/"* - [from ArgoUml]
>
> *"TODO: This dialect is not yet complete. Need to provide implementations wherever Not yet implemented appears"* - [from SQuirrel]

To mitigate the risk of creating a dataset that is biased, we extracted a statistically significant sample of our dataset and asked another student to classify it. To prepare the student for the task we gave a 1-hour tutorial about the different kinds of self-admitted technical debt, and walked the student through a couple of examples of each different type of self-admitted technical debt comment. The tutorial is provided online as well [26]. The statistically significant sample was created based on the total number of comments (62,566) with a confidence level of 99 percent and a confidence interval of 5 percent, resulting in a stratified sample of 659 comments. We composed the stratified sample according to the percentage of each classification found in the original dataset. Therefore, the stratified sample was composed of: 92 percent comments without self-admitted technical debt (609 comments), 4 percent design debt (29 comments), 2 percent requirement debt (5 comments), 0.75 percent test debt (2 comments) and 0.15 percent documentation debt (1 comment). Lastly, we evaluate the level of agreement between both reviewers of the stratified sample by calculating Cohen's kappa coefficient [31]. The Cohen's Kappa coefficient has been commonly used to evaluate inter-rater agreement level for categorical scales, and provides the proportion of agreement corrected for chance. The resulting coefficient is scaled to range between -1 and +1, where a negative value means poorer than chance agreement, zero indicates exactly chance agreement, and a positive value indicates better than chance agreement [32]. The closer the value is to +1, the stronger the agreement. In our work, the level of agreement measured between the reviewers was of +0.81.

We also measured the level of agreement in the classification of design and requirement self-admitted technical debt individually. This is important because the stratified sample contains many more comments without self-admitted technical debt than the other types of debt, and therefore, the coefficient reported above could indicate that the reviewers are agreeing on what is not self-admitted technical debt, instead of agreeing on a particular type of debt. However, we achieved a level of agreement of +0.75 for design self-admitted technical debt, and +0.84 for requirement self-admitted technical debt. According to Fleiss [33] values larger than +0.75 are characterized as excellent agreement.

2.5 NLP Classification

Our next step is to use the classified self-admitted technical debt comments as a training dataset for the Stanford Classifier, which is a Java implementation of a maximum entropy classifier [25]. A maximum entropy classifier, in general, takes as input a number of data items along with a classification for each data item, and automatically generates *features* (i.e., words) from each *datum*, which are associated with positive or negative numeric *votes* for each class. The weights of the features are learned automatically based on the manually classified training data items (supervised learning). The Stanford Classifier builds a *maximum entropy model*, which is equivalent to a multi-class regression model, and it is trained to maximize the conditional likelihood of the classes taking into account feature dependences when calculating the feature weights.

After the training phase, the maximum entropy classifier can take as input a test dataset that will be classified according to the model built during the training phase. The output for each data item in the test dataset is a classification, along with the features contributing positively or negatively in this classification.

In our case, the training dataset is composed of source code comments and their corresponding manual classification. According to our findings in previous work [15], the two most common types of self-admitted technical debt are design and requirement debt (defect, test, and documentation debt together represent less that 10 percent of all self-admitted technical debt comments). Therefore, we train the-maximum entropy classifier on the dataset containing only these two specific types of self-admitted technical debt comments.

In order to avoid having repeated features differing only in letter case (e.g., “Hack”, “hack”, “HACK”), or in preceding/succeeding punctuation characters (e.g., “,hack”, “hack,”), we preprocess the training and test datasets to clean up the original comments written by the developers. More specifically, we remove the character structures that are used in the Java language syntax to indicate comments (i.e., ‘//’ or ‘/*’ and ‘*/’), the punctuation characters (i.e., ‘,’, ‘...’, ‘;’, ‘:’), and any excess whitespace characters (e.g., ‘ ’, ‘\t’, ‘\n’), and finally we convert all comments to lowercase. However, we decided not to remove exclamation and interrogation marks. These specific punctuations were very useful during the identification of self-admitted technical debt comments, and provide insightful information about the meaning of the features.

3 Experiment Results

The goal of our research is to develop an automatic way to detect design and requirement self-admitted technical debt comments. To do so, we first manually classify a large number of comments identifying those containing self-admitted technical debt. With the resulting dataset, we train the maximum entropy classifier to identify design and requirement self-admitted technical debt (RQ1). To better understand what words indicate self-admitted technical debt, we inspect the features used by the maximum entropy classifier to identify the detected self-admitted technical debt. These features are words that are frequently found in comments with technical debt. We present the 10 most common words that indicate design and requirement self-admitted technical debt (RQ2). Since the manual classification required to create our training dataset is expensive, ideally we would like to achieve maximum performance with the least amount of training data. Therefore, we investigate how variations in the size of training data affects the performance of our classification (RQ3). We detail the motivation, approach and present the results of each of our research questions in the remainder of this section.

RQ1. Is it possible to more accurately detect self-admitted technical debt using NLP techniques?

Motivation. As shown in previous work [15], self-admitted technical debt comments can be found in the source code. However, there is no automatic way to identify these comments. The methods proposed so far heavily rely on the manual inspection of source code, and there is no evidence on how well these approaches perform. Moreover, most of them do not discriminate between the different types of technical debt (e.g., design, test, requirement).

Therefore, we want to determine if NLP techniques such as, the maximum entropy classifier, can help us surpass these limitations and outperform the accuracy of the current state-of-the-art. The maximum entropy classifier can automatically classify comments based on specific linguistic characteristics of these comments. Answering this question is important, since it helps us understand the opportunities and limitations of using NLP techniques to automatically identify self-admitted technical debt comments.

Approach. For this research question, we would like to examine how effectively we can identify design and requirement self-admitted technical debt. Therefore, the first step is to create a dataset that we can train and test the maximum entropy classifier on. We classified the source code comments into the following types of self-admitted technical debt: design, defect, documentation, requirement, and test debt. However, our previous work showed that the most frequent self-admitted technical debt comments are design and requirement debt. Therefore, in this paper, we focus on the identification of these two types of self-admitted technical debt, because 1) they are the most common types of technical debt, and 2) NLP-based techniques require sufficient data for training (i.e., they cannot build an accurate model with a small number of samples).

We train the maximum entropy classifier using our manually created dataset. The dataset contains comments with and without self-admitted technical debt, and each comment has a classification (i.e., without technical debt, design debt, or requirement debt). Then, we add to the training dataset all comments classified as without technical debt and the comments classified as the specific type of self-admitted technical debt that we want to identify (i.e., design or requirement debt). We use the comments from 9 out of the 10 projects that we analyzed to create the training dataset. The comments from the remaining one project are used

TABLE 2
Comparison of F1-Measure Between the NLP-Based, the Comment Patterns and the Random Baseline Approaches for Design and Requirement Debt

Project	Design Debt					Requirement Debt				
	Our Approach	Comment Patterns	Random Classifier	Imp. Over Comment Patterns	Imp. Over Random Classifier	Our Approach	Comment Patterns	Random Classifier	Imp. Over Comment Patterns	Imp. Over Random Classifier
Ant	0.517	0.237	0.044	2.1×	11.7×	0.154	0.000	0.006	-	25.6×
ArgoUML	0.814	0.107	0.144	7.6×	5.6×	0.595	0.000	0.079	-	7.5×
Columba	0.601	0.264	0.037	2.2×	16.2×	0.804	0.117	0.013	6.8×	61.8×
EMF	0.470	0.231	0.034	2.0×	13.8×	0.381	0.000	0.007	-	54.4×
Hibernate	0.744	0.227	0.193	3.2×	3.8×	0.476	0.000	0.041	-	11.6×
JEdit	0.509	0.342	0.037	1.4×	13.7×	0.091	0.000	0.003	-	30.3×
JFreeChart	0.492	0.282	0.077	1.7×	6.3×	0.321	0.000	0.007	-	45.8×
JMeter	0.731	0.194	0.072	3.7×	10.1×	0.237	0.148	0.005	1.6×	47.4×
JRuby	0.783	0.620	0.123	1.2×	6.3×	0.435	0.409	0.043	1.0×	10.1×
SQuirrel	0.540	0.175	0.055	3.0×	9.8×	0.541	0.000	0.014	-	38.6×
Average	0.620	0.267	0.081	2.3×	7.6×	0.403	0.067	0.021	6.0×	19.1×

to evaluate the classification performed by the maximum entropy classifier. We choose to create the training dataset using comments from 9 out of 10 projects, because we want to train the maximum entropy classifier with the most diverse data possible (i.e., comments from different domains of applications). However, we discuss the implications of using training datasets of different sizes in RQ3. We repeat this process for each one of the ten projects, each time training on the other 9 projects and testing on the remaining 1 project.

Based on the training dataset, the maximum entropy classifier will classify each comment in the test dataset. The resulting classification is compared with the manual classification provided in the test dataset. If a comment in the test dataset has the same manual classification as the classification suggested by the maximum entropy classifier, we will have a true positive (tp) or a true negative (tn). True positives are the cases where the maximum entropy classifier correctly identifies self-admitted technical debt comments, and true negatives are comments without technical debt that are classified as being as such. Similarly, when the classification provided by the tool diverges from the manual classification provided in the test dataset, we have false positives or false negatives. False positives (fp) are comments classified as being self-admitted technical debt when they are not, and false negatives (fn) are comments classified as without technical debt when they really are self-admitted technical debt comments. Using the tp, tn, fp, and fn values, we are able to evaluate the performance of different detection approaches in terms of precision ($P = \frac{tp}{tp+fp}$), recall ($R = \frac{tp}{tp+fn}$) and F1-measure ($F = 2 \times \frac{P \times R}{P+R}$). To determine how effective the NLP classification is, we compare its F1-measure values with the corresponding F1-measure values of the two other approaches. We use the F1-measure to compare the performance between the approaches as it is the harmonic mean of precision and recall. Using the F1-measure allows us to incorporate the trade-off between precision and recall and present one value that evaluates both measures.

The first approach is the current state-of-the-art in detecting self-admitted technical debt comments [14]. This approach uses 62 comment patterns (i.e., keywords and phrases) that were found as recurrent in self-admitted technical debt comments during the manual inspection of 101,762 comments. The second approach is a simple (random) baseline, which assumes that the detection of self-admitted technical debt is random (this approach is used as a F1 lower bound). The precision of this approach is calculated by taking the total number of self-admitted technical debt over the total number of comments of each project. For example, project Ant has 4,137 comments, of those, only 95 comments are design self-admitted technical debt. The probability of randomly labelling a comment as a self-admitted technical debt comment is 0.023 (i.e., $\frac{95}{4{,}137}$). Similarly, to calculate the recall we take into consideration the two possible classifications available: one is the type of self-admitted technical debt (e.g., design) and the other is without technical debt. Therefore, there is a 50 percent chance that the comment will be classified as self-admitted technical debt. Thus, the F1-measure for the random baseline for project Ant is computed as $2 \times \frac{0.023 \times 0.5}{0.023+0.5} = 0.044$.

Results—Design Debt. Table 2 presents the F1-measure of the three approaches, as well as the improvement achieved by our approach compared to the other two approaches. We see that for all projects, the F1-measure achieved by our approach is higher than the other approaches. The F1-measure values obtained by our NLP-based approach range between 0.470-0.814, with an average of 0.620. In comparison, the F1-measure values using the comment patterns range between 0.107-0.620, with an average of 0.267, while the simple (random) baseline approach achieves F1-measure values in the range of 0.034-0.193, with an average of 0.081. Fig. 2a visualizes the comparison of the F1-measure values for our NLP-based approach, the comment patterns approach, and the simple (random) baseline approach. We see from both, Table 2 and Fig. 2a that, on average, our approach outperforms the comment patterns approach by 2.3 times and the simple (random) baseline approach by 7.6 times when identifying design self-admitted technical debt.

It is important to note that the comment patterns approach has a high precision, but low recall, i.e., this approach points correctly to self-admitted technical debt comments, but as it depends on keywords, it identifies a very small subset of all the self-admitted technical debt

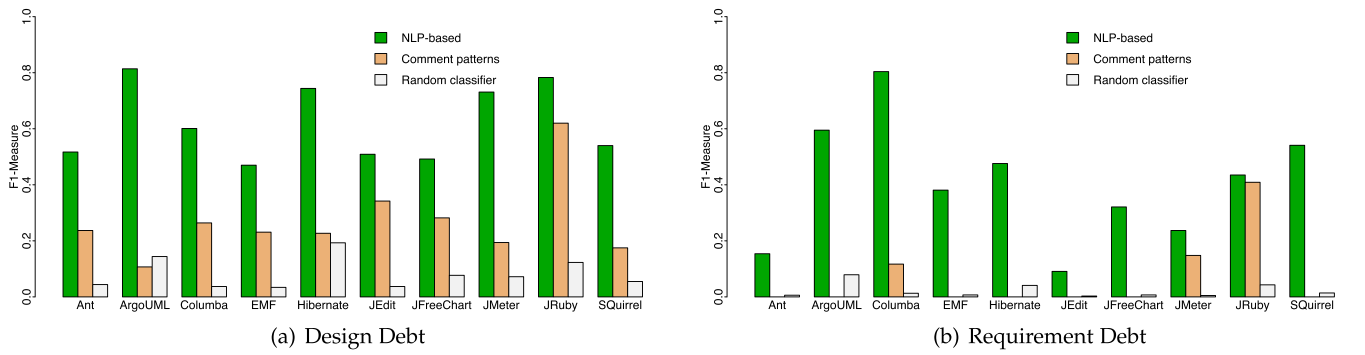

Fig. 2. Visualization of the F1-measure for different approaches.

comments in the project. Although we only show the F1-measure values here, we present the precision and recall values in Table 9 in the Appendix section.

Results—Requirement Debt. Similarly, the last five columns of Table 2 show the F1-measure performance of the three approaches, and the improvement achieved by our approach over the two other approaches. The comment patterns approach was able to identify requirement self-admitted technical debt in only 3 of the 10 analyzed projects. A possible reason for the low performance of the comment patterns in detecting requirement debt is that the comment patterns do not differentiate between the different types of self-admitted technical debt. Moreover, since most of the debt is design debt, it is possible that the patterns tend to favor the detection of design debt.

That said, we find that for all projects, the F1-measure values obtained by our approach surpass the F1-measure values of the other approaches. Our approach achieves F1-measure values between 0.091-0.804 with an average of 0.403, whereas the comment pattern approach achieves F1-measure values in the range of 0.117-0.409 with an average of 0.067, while the simple (random) baseline ranges between 0.003-0.079, with an average of 0.021. Fig. 2b visualizes the performance comparison of the two approaches. We also examine if the differences in the F1-measure values obtained by our approach and the other two baselines are statistically significant. Indeed, we find that the differences are statistically significant ($p<0.001$) for both baselines and both design and requirement self-admitted technical debt.

Generally, requirement self-admitted technical debt is less common than design self-admitted technical debt, which makes it more difficult to detect. Nevertheless, our NLP-based approach provides a significant improvement over the comment patterns approach, outperforming it by 6 times, on average. Table 2 only presents the F1-measure values for the sake of brevity, however, we present the detailed precision and recall values in the Appendix section, Table 10.

We find that our NLP-based approach, is more accurate in identifying self-admitted technical debt comments compared to the current state-of-art. We achieved an average F1-measure of 0.620 when identifying design debt (an average improvement of 2.3× over the state-of-the-art approach) and an average F1-measure of 0.403 when identifying requirement debt (an average improvement of 6× over the state-of-the-art approach).

RQ2. What are the most impactful words in the classification of self-admitted technical debt?

Motivation. After assessing the accuracy of our NLP-based approach in identifying self-admitted technical debt comments, we want to better understand what words developers use when expressing technical debt. Answering this question will provide insightful information that can guide future research directions, broaden our understanding on self-admitted technical debt and also help us to detect it.

Approach. The maximum entropy classifier learns optimal features that can be used to detect self-admitted technical debt. A feature is comment fragment (e.g., word) that is associated with a specific class (i.e., design debt, requirement debt, or without technical debt), and a weight that represents how strongly this feature relates to that class. The maximum entropy classifier uses the classified training data to determine the features and their weights. Then, these features and their corresponding weights are used to determine if a comment belongs to a specific type of self-admitted technical debt or not.

For example, let us assume that after the training, the maximum entropy classifier determines that the features "hack" and "dirty" are related to the *design-debt* class with weights 5.3 and 3.2, respectively, and the feature "something" relates to the *without-technical-debt* class with a weight of 4.1. Then, to classify the comment "this is a dirty hack it's better to do something" from our test data, all features present in the comment will be examined and the following scores would be calculated: $weight_{design\text{-}debt} = 8.5$ (i.e., the sum of "hack" and "dirty" feature weights) and $weight_{without\text{-}technical\text{-}debt} = 4.1$. Since $weight_{design\text{-}debt}$ is larger than $weight_{without\text{-}technical\text{-}debt}$, the comment will be classified as design debt.

For each analyzed project, we collect the features used to identify the self-admitted technical debt comments. These features are provided by the maximum entropy classifier as output and stored in a text file. The features are written in the file according to their weights in descending order (starting from more relevant, ending to less relevant features). Based on these files, we rank the words calculating the average ranking position of the analyzed features across the ten different projects.

Results. Table 3 shows the top-10 textual features used to identify self-admitted technical debt in the ten studied projects, ordered by their average ranking. The first column shows the ranking of each textual feature, the second column lists the features used in the identification of *design*

TABLE 3
Top-10 Textual Features Used to Identify Design and Requirement Self-Admitted Technical Debt

Rank	Design Debt	Requirement Debt
1	**hack**	**todo**
2	**workaround**	**needed**
3	yuck!	**implementation**
4	kludge	**fixme**
5	stupidity	**xxx**
6	needed?	ends?
7	columns?	convention
8	unused?	configurable
9	wtf?	apparently
10	todo	fudging

self-admitted technical debt, and the third column lists the textual features used to identify *requirement* self-admitted technical debt.

From Table 3 we observe that the top ranked textual features for design self-admitted technical debt, i.e., *hack*, *workaround*, *yuck!*, *kludge* and *stupidity*, indicate sloppy code, or mediocre source code quality. For example, we have the following comment that was found in JMeter:

"Hack to allow entire URL to be provided in host field"

Other textual features, such as *needed?*, *unused?* and *wtf?* are questioning the usefulness or utility of a specific source code fragment, as indicated by the following comment also found in JMeter:

"TODO: - is this needed?"

For requirement self-admitted technical debt, the top ranked features, i.e., *todo*, *needed*, *implementation*, *fixme* and *xxx* indicate the need to complete requirements in the future that are currently partially complete. An indicative example is the following one found in JRuby:

"TODO: implement, won't do this now"

Some of the remaining lower ranked textual features, such as *convention*, *configurable* and *fudging* also indicate potential incomplete requirements, as shown in the following comments:

"Need to calculate this... just fudging here for now" [from JEdit]

"could make this configurable" [from JFreeChart]

"TODO: This name of the expression language should be configurable by the user" [from ArgoUML]

"TODO: find a way to check the manifest-file, that is found by naming convention" [from Apache Ant]

It should be noted that the features highlighted in bold in Table 3 appear in all top-10 lists extracted from each one of the ten training datasets, and therefore can be considered as more *universal/stable* features compared to the others.

We also observe that it is possible for a single textual feature to indicate both design and requirement self-admitted technical debt. However, in such cases, the ranking of the feature is different for each kind of debt. For example, the word "todo" is ranked tenth for design debt, whereas it is ranked first for requirement debt. This finding is intuitive, since requirement debt will naturally be related to the implementation of future functionality.

It is important to note here that although we present only the top-10 textual features, the classification of the comments is based on a combination of a large number of textual features. In fact, two different types of textual features are used to classify the comments, namely positive and negative weight features. Positive weight features will increase the total weight of the vote suggesting that the classification should be equal to the class of the feature (i.e., design or requirement debt). On the other hand, negative weight features will decrease the total weight of the vote suggesting a classification different from the class of the feature. On average, the number of positive weight features used to classify design and requirement debt is 5,014 and 2,195, respectively. The exact number of unique textual features used to detect self-admitted technical debt for each project is shown in Table 4. The fact that our NLP-based approach leverages so many features helps to explain the significant improvement we are able to achieve over the state-of-the-art, which only uses 62 patterns. In comparison, our approach leverages 35,828 and 34,056 unique textual features for detecting comments with design and requirement debt, respectively.

TABLE 4
Number of Unique Textual Features Use to Detect Design and Requirement Debt for Each Project

Project	Design Debt			Requirement Debt		
	Positive Weight Features	Negative Weight Features	# of Features	Positive Weight Features	Negative Weight Features	# of Features
Ant	5,299	23,623	28,922	1,812	27,673	29,485
ArgoUML	3,917	26,012	29,929	2,779	27,260	30,039
Columba	5,255	24,182	29,437	2,433	27,561	29,994
EMF	5,346	23,667	29,013	1,889	27,637	29,526
Hibernate	4,914	24,070	28,984	2,748	26,654	29,402
JEdit	5,042	24,644	29,686	1,831	28,267	30,098
JFreeChart	5,361	23,530	28,891	1,902	27,439	29,341
JMeter	5,172	23,916	29,088	1,893	27,716	29,609
JRuby	4,856	24,553	29,409	2,850	27,085	29,935
SQuirrel	4,982	25,146	30,128	1,814	26,914	28,728
Average	5,014	24,334	29,348	2,195	27,420	29,615
Total unique	6,327	31,518	35,828	4,015	32,954	34,056

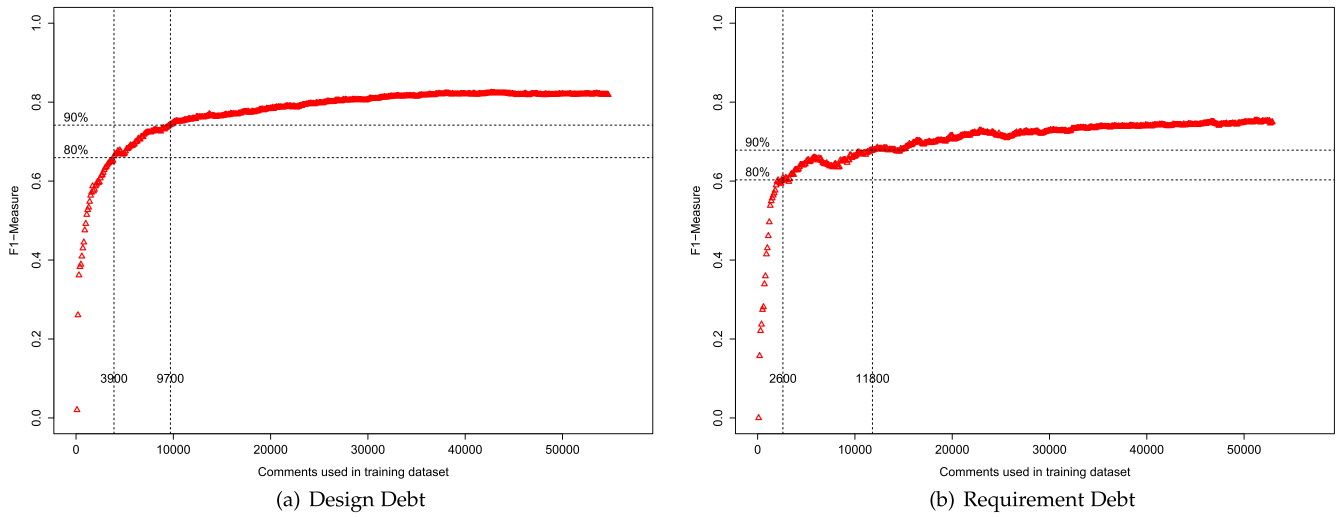

Fig. 3. F1-measure achieved by incrementally adding batches of 100 comments in the training dataset.

We find that design and requirement debt have their own textual features that best indicate such self-admitted technical debt comments. For design debt, the top textual features indicate sloppy code or mediocre code quality, whereas for requirement debt they indicate the need to complete a partially implemented requirement in the future.

RQ3. How much training data is required to effectively detect self-admitted technical debt?

Motivation. Thus far, we have shown that our NLP-based approach can effectively identify comments expressing self-admitted technical debt. However, we conjecture that the performance of the classification depends on the amount of training data. At the same time, creating the training dataset is a time consuming and labor intensive task. So, the question that arises is: how much training data do we need to effectively classify the source code comments? If we need a very large number of comments to create our training dataset, our approach will be more difficult to extend and apply for other projects. On the other hand, if a small dataset can be used to reliably identify comments with self-admitted technical debt, then this approach can be applied with minimal effort, i.e., less training data. That said, intuitively we expect that the performance of the maximum entropy classifier will improve as more comments are being added to the training dataset.

Approach. To answer this research question, we follow a systematic process where we incrementally add training data and evaluate the performance of the classification. More specifically, we combine the comments from all projects into a single large dataset. Then, we split this dataset into ten equally-sized folds, making sure that each partition has the same ratio of comments of self-admitted technical debt and without technical debt as the original dataset. Next, we use one of the ten folds for testing and the remaining nine folds as training data. Since we want to examine the impact of the quantity of training data on performance, we train the classifier with batches of 100 comments at a time and test its performance on the testing data. It is important to note that even within the batches of 100 comments, we maintain the same ratio of self-admitted technical debt and non technical debt comments as in the original dataset. We keep adding comments until all of the training dataset is used. We repeat this process for each one of the ten folds and report the average performance across all folds.

We compute the F1-measure values after each iteration (i.e., the addition of a batch of 100 comments) and record the iteration that achieves the highest F1-measure. Then we find the iterations in which at least 80 and 90 percent of the maximum F1-measure value is achieved, and report the number of comments added up to those iterations.

Results—Design Debt. Fig. 3a shows the average F1-measure values obtained when detecting design self-admitted technical debt, while adding batches of 100 comments. We find that the F1-measure score improves as we increase the number of comments in the training dataset, and the highest value (i.e., 0.824) is achieved with 42,700 comments. However, the steepest improvement in the F1-measure performance takes place within the first 2K-4K comments. Additionally, 80 and 90 percent of the maximum F1-measure value is achieved with 3,900 and 9,700 comments in the training dataset, respectively. Since each batch of comments consists of approximately 5 percent (i.e., $\frac{2{,}703}{58{,}122}$) comments with design self-admitted technical debt, the iteration achieving 80 percent of the maximum F1-measure value contains 195 comments with design self-admitted technical debt, while the iteration achieving 90 percent of the maximum F1-measure value contains 485 such comments. In conclusion, to achieve 80 percent of the maximum F1-measure value, we need only 9.1 percent (i.e., $\frac{3{,}900}{42{,}700}$) of the training data, while to achieve 90 percent of the maximum F1-measure value, we need only 22.7 percent (i.e., $\frac{9{,}700}{42{,}700}$) of the training data.

Results—Requirement Debt. Fig. 3b shows the average F1-measure values obtained when detecting requirement self-admitted technical debt, while adding batches of 100 comments. As expected, the F1-measure increases as we add more comments into the training dataset, and again the steepest improvement takes place within the first 2-3K comments. The highest F1-measure value (i.e., 0.753) is achieved using 51,300 comments of which 675 are requirement self-admitted technical debt. Additionally, 80 percent of the maximum F1-measure score is achieved with 2,600 comments, while 90 percent of the maximum F1-measure score with 11,800 comments in the training dataset. Each batch contains two comments with requirement self-admitted technical debt, since the percentage of such comments is 1.3 percent (i.e., $\frac{757}{58,122}$) in the entire dataset. As a result, the iteration achieving 80 percent of the maximum F1-measure value contains 52 comments with requirement self-admitted technical debt, while the iteration achieving 90 percent of the maximum F1-measure value contains 236 such comments. In conclusion, to achieve 80 percent of the maximum F1-measure value, we need only 5 percent (i.e., $\frac{2,600}{51,300}$) of the training data, while to achieve 90 percent of the maximum F1-measure value, we need only 23 percent (i.e., $\frac{11,800}{51,300}$) of the training data.

We find that to achieve a performance equivalent to 90 percent of the maximum F1-measure score, only 23 percent of the comments are required for both design and requirement self-admitted technical debt. For a performance equivalent to 80 percent of the maximum F1-measure score, only 9 and 5 percent of the comments are required for design and requirement self-admitted technical debt, respectively.

4 Discussion

Thus far, we have seen that our NLP-based approach can perform well in classifying self-admitted technical debt. However, there are some observations that warrant further investigation. For example, when it comes to the different types of self-admitted technical debt, we find that requirement debt tends to require less training data, which is another interesting point that is worth further investigation (Section 4.1).

Moreover, we think that is also interesting to know the performance of our approach when trained to distinguish between self-admitted technical debt and non-self-admitted technical debt, i.e., without using fine-grained classes of debt, such as design and requirement debt (Section 4.2).

Also, when performing our classification, there are several different classifiers that can be used in the Stanford Classifier toolkit, hence we investigate what is the impact of using different classifiers on the accuracy (Section 4.3).

Lastly, we analyze the overlap between the files that contain self-admitted technical debt and the files that contain code smells. This is an interesting point of discussion to provide insights on how technical debt found in comments relates to code smells found by static analysis tools (Section 4.4).

4.1 Textual Similarity for Design and Requirement Debt

For RQ3, we hypothesize that one of the reasons that the detection of requirement self-admitted technical debt comments needs less training data is because such comments are more similar to each other compared to design self-admitted technical debt comments. Therefore, we compare the intra-similarity of the requirement and design debt comments.

We start by calculating the term frequency-inverse document frequency (*tf-idf*) weight of each design and requirement self-admitted technical debt comment. Term frequency (*tf*) is the simple count of occurrences that a term

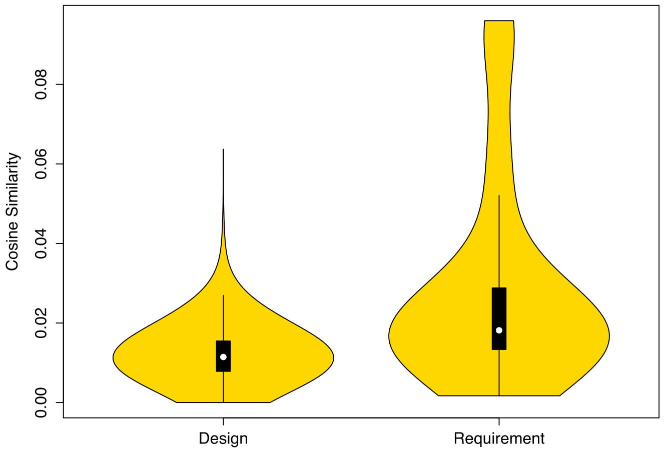

Fig. 4. Textual similarity between design and requirement debt comments.

(i.e., word) has in a document (i.e., comment). Inverse document frequency (*idf*) takes into account the number of documents that the term appears. However, as the name implies, the more one term is repeated across multiple documents the less relevant it is. Therefore, let N be the total number of documents in a collection, the *idf* of a term t is defined as follows: $idf_t = \log \frac{N}{df_t}$. The total *tf-idf* weight of a document is equal to the sum of each individual term *tf-idf* weight in the document. Each document is represented by a *document vector* in a *vector space model*.

Once we have the *tf-idf* weights for the comments, we calculate the *cosine similarity* between the comments. The Cosine similarity can be viewed as the *dot product* of the normalized versions of two document vectors (i.e., two comments) [34]. The value of the cosine distance ranges between 0 to 1, where 0 means that the comments are not similar at all and 1 means that the comments are identical.

For example, the requirement self-admitted technical debt dataset contains 757 comments, for which we generate a 757×757 matrix (since we compare each comment to all other comments). Finally, we take the average cosine similarity for design and requirement debt comments, respectively, and plot their distributions. Fig. 4 shows that the median and the upper quartile for requirement self-admitted technical debt comments are higher than the median and upper quartile for design self-admitted technical debt. The median for requirement debt comments is 0.018, whereas, the median for design debt comments is 0.011. To ensure that the difference is statistically significant, we perform the Wilcoxon test to calculate the p-value. The calculated p-value is less than 2.2e-16 showing that the result is indeed statistically significant (i.e., $p < 0.001$). Considering our findings, our hypothesis is validated, showing that requirement self-admitted technical debt comments are more similar to each other compared to design self-admitted technical debt comments. This may help explain why requirement debt needs a smaller set of positive weight textual features to be detected.

4.2 Distinguishing Self-Admitted Technical Debt from Non-Self-Admitted Technical Debt Comments

So far, we analyzed the performance of our NLP-based approach to identify distinct types of self-admitted technical

TABLE 5
F1-Measure Performance Considering Different Types of Self-Admitted Technical Debt

Project	Design Debt	Requirement Debt	Technical Debt
Ant	0.517	0.154	0.512
ArgoUML	0.814	0.595	0.819
Columba	0.601	0.804	0.750
EMF	0.470	0.381	0.462
Hibernate	0.744	0.476	0.763
JEdit	0.509	0.091	0.461
JFreeChart	0.492	0.321	0.513
JMeter	0.731	0.237	0.715
JRuby	0.783	0.435	0.773
SQuirrel	0.540	0.541	0.593
Average	0.620	0.403	0.636

TABLE 6
Top-10 Textual Features Used to Identify Different Types of Self-Admitted Technical Debt

Project	Design Debt	Requirement Debt	Technical Debt
1	hack	todo	hack
2	workaround	needed	workaround
3	yuck!	implementation	yuck!
4	kludge	fixme	kludge
5	stupidity	xxx	stupidity
6	needed?	ends?	needed?
7	columns?	convention	unused?
8	unused?	configurable	fixme
9	wtf?	apparently	todo
10	todo	fudging	wtf?

debt (i.e., design and requirement debt). However, a simpler distinction between self-admitted technical debt and non-debt comments can also be interesting in the case that those fine-grained classes of debt are not considered necessary by a user of the proposed NLP-based detection approach. Another reason justifying such a coarse-grained distinction is that the cost of building a training dataset with fine-grained classes of debt is more expensive, mentally challenging, and subjective than building a training dataset with just two classes (i.e., comments with and without technical debt).

In order to compute the performance of our NLP-based approach using only two classes (i.e., comments with and without technical debt), we repeat RQ1 and RQ2 with modified training and test datasets. First, we take all design and requirement self-admitted technical debt comments and label them with a common class i.e., technical debt, and the remaining comments we kept them labeled as without technical debt. Second, we run the maximum entropy classifier in the same leave-one-out cross-project validation fashion, using the comments of 9 projects to train the classifier and the comments from the remaining project to test the classifier. We repeat this process for each of the ten projects and compute the average F1-measure. Lastly, we analyze the textual features used to identify the self-admitted technical debt comments.

Table 5 compares the F1-measure achieved when detecting design debt, requirement debt, separately and when detecting both combined in a single class. As we can see, the performance when detecting technical debt is very similar with the performance of the classifier when detecting design debt. This is expected, as the majority of technical debt comments in the training dataset are labeled with the design debt class. Nevertheless, the performance achieved when detecting design debt was surpassed in the projects where the classifier performed well in detecting requirement debt, for example, in Columba (0.601 versus 0.750) and SQuirrel SQL (0.540 versus 0.593).

We find that the average performance when detecting design and requirement self-admitted technical debt combined is better (0.636) than the performance achieved when detecting them individually (0.620 and 0.403 for design and requirement debt, respectively).

Table 6 shows a comparison of the top-10 textual features used to detect design and requirement debt comments separately, and those used to detect both types of debt combined in a single class. When analyzing the top-10 textual features used to classify self-admitted technical debt, we find once more, a strong overlap with the top-10 textual features used to classify design debt. The weight of the features is attributed in accordance to the frequency that each word is found in the training dataset, and therefore, the top-10 features tend to be similar with the top-10 design debt features, since design debt comments represent the majority of self-admitted technical debt comments in the dataset.

4.3 Investigating the Impact of Different Classifiers on the Accuracy of the Classification

In our work, the classification performed by the Stanford Classifier used the maximum entropy classifier. However, the Stanford Classifier can use other classifiers too. In order to examine the impact of the underlying classifier on the accuracy of the proposed approach, we investigate two more classifiers, namely the Naive Bayes, and the Binary classifiers.

Figs. 5a and 5b compare the performance between the three different classifiers. We find that the Naive Bayes has the worst average F1-measure of 0.30 and 0.05 for design and requirement technical debt, respectively. Based on our findings, the Naive Bayes algorithm favours recall at the expense of precision. For example, while classifying design debt, the average recall was 0.84 and precision 0.19. The two other algorithms present more balanced results compared to the Naive Bayes, and the difference in their performance is almost negligible. The Logistic Regression classifier achieved F1-measures of 0.62 and 0.40, while the Binary classifier F1-measures were 0.63 and 0.40, for design and requirement self-admitted technical debt, respectively. Tables 11 and 12 in the Appendix section provide detailed data for each classifier and all ten examined projects.

Although the Binary classifier has a slightly better performance, for our purpose, the Logistical Regression classifier provides more insightful textual features. These features were analyzed and presented in RQ2.

According to previous work, developers hate to deal with false positives (i.e., low precision) [35], [36], [37]. Due to this fact, we choose to present our results in this study using the maximum entropy classifier, which has an average precision of 0.716 throughout all projects. However, favouring recall over precision by using the Naives Bayes classifier might still be acceptable, if a manual process to

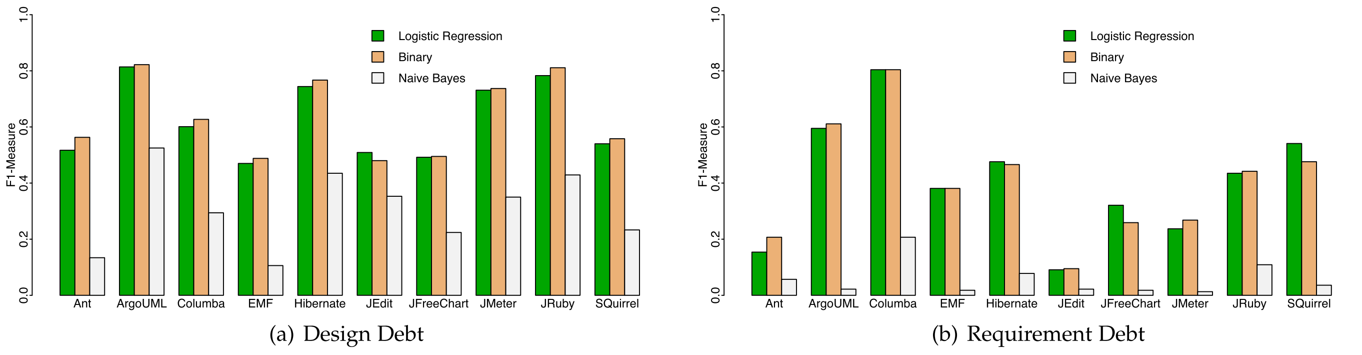

Fig. 5. Underlying classifier algorithms performance comparison.

filter out false positives is in place, as reported by Berry et al. [38].

One important question to ask when choosing what kind of classifier to use is how much training data is currently available. In most of the cases, the trickiest part of applying a machine learning classifier in real world applications is creating or obtaining enough training data. If you have fairly little data at your disposal, and you are going to train a supervised classifier, then machine learning theory recommends classifiers with high bias, such as the Naive Bayes [39], [40]. If there is a reasonable amount of labeled data, then you are in good stand to use most kinds of classifiers [34]. For instance, you may wish to use a Support Vector Machine (SVM), a decision tree or, like in our study, a max entropy classifier. If a large amount of data is available, then the choice of classifier probably has little effect on the results and the best choice may be unclear [41]. It may be best to choose a classifier based on the scalability of training, or even runtime efficiency.

4.4 Investigating the Overlap Between Technical Debt Found in Comments and Technical Debt Found by Static Analysis Tools

Thus far, we analyzed technical debt that was expressed by developers through source code comments. However, there are other ways to identify technical debt, such as architectural reviews, documentation analysis, and static analysis tools. To date, using static analysis tools is one of the most popular approaches to identify technical debt in the source code [42]. In general, static analysis tools parse the source code of a project to calculate metrics and identify possible object oriented design violations, also known as code smells, anti-patterns, or design technical debt, based on some fixed metric threshold values.

We analyze the overlap between what our NLP-based approach identifies as technical debt and what a static analysis tool identifies as technical debt. We selected JDeodorant as the static analysis tool, since it supports the detection of three popular code smells, namely Long Method, God Class, and Feature Envy. We avoided the use of metric-based code smell detection tools, because they tend to have high false positive rates and flag a large portion of the code base as problematic [23]. On the other hand, JDeodorant detects only actionable code smells (i.e., code smells for which a behavior-preserving refactoring can be applied to resolve them), and does not rely on any metric thresholds, but rather applies static source code analysis to detect structural anomalies and suggest refactoring opportunities to eliminate them [29].

First, we analyzed our 10 open source projects using JDeodorant. The result of this analysis is a list of Java files that were identified having at least one instance of the Long Method, God Class, and Feature Envy code smells. Table 7 shows the total number of files and the number/percentage of files that contain each of the aforementioned smells. We find that, on average, 29.3, 5.5, 16.6 percent of all files have at least one instance of the Long Method, Feature Envy or God Class smells, respectively. These code smells have been

TABLE 7
Detailed Information About the Files Containing Bad Smells as Detected by JDeodorant

Project	# of files	# of Files with Long Method	% of Files with Long Method	# of Files with Feature Envy	% of Files with Feature Envy	# of Files with God Class	% of Files with God Class	# of Files with Any Code Smell	% of Files with Any Code Smell
Ant	1,475	508	34.4	110	7.4	365	24.7	612	41.4
ArgoUML	2,609	654	25.0	62	2.3	249	9.5	730	27.9
Columba	1,711	505	29.5	65	3.7	244	14.2	593	34.6
EMF	1,458	362	24.8	50	3.4	231	15.8	448	30.7
Hibernate	1,356	216	15.9	69	5.0	190	14.0	331	24.4
JEdit	800	268	33.5	57	7.1	133	16.6	311	38.8
JFreeChart	1,065	523	49.1	54	5.0	231	21.6	583	54.7
JMeter	1,181	487	41.2	113	9.5	241	20.4	564	47.7
JRuby	1,486	319	21.4	87	5.8	218	14.6	394	26.5
SQuirrel	3,108	566	18.2	204	6.5	466	14.9	825	26.5
Average			29.3		5.5		16.6		35.3

TABLE 8
Overlap Between the Files Containing Self-Admitted Debt and the Files Containing Code Smells as Detected by JDeodorant

Project	# of Files with SATD	# of SATD Files with Long Method	% of SATD Files with Long Method	# of SATD Files with Feature Envy	% of SATD Fles with Feature Envy	# of SATD Files with God Class	% of SATD Files with God Class	# of SATD Files with Any Code Smell	% of SATD Files with Any Code Smell
Ant	73	57	78.0	19	26.0	42	57.5	63	86.3
ArgoUML	419	255	60.8	43	10.2	128	30.5	283	67.5
Columba	117	76	64.9	18	15.3	47	40.1	89	76.0
EMF	53	33	62.2	14	26.4	28	52.8	28	52.8
Hibernate	206	90	43.6	44	21.3	72	34.9	116	56.3
JEdit	108	74	68.5	23	21.2	47	43.5	82	75.9
JFreeChart	106	87	82.0	20	18.8	52	49.0	92	86.7
JMeter	200	143	71.5	41	20.5	97	48.5	161	80.5
JRuby	163	107	65.5	43	26.3	79	48.4	85	52.1
SQuirrel	156	82	52.5	32	20.5	58	37.1	99	63.4
Average			65.0		20.7		44.2		69.7

extensively investigated in the literature, and are considered to occur frequently [43], [44]. Second, we created a similar list containing the files that were identified with self-admitted technical debt comments. Finally, we examined the overlap of the two lists of files. It should be emphasized that we did not examine if the self-admitted technical debt comments actually discuss the detected code smells, but only if there is a co-occurrence at file-level.

Table 8 provides details about each one of the projects used in our study. The columns of Table 8 present the total number of files with self-admitted technical debt, followed by the number of files containing self-admitted technical debt comments and at least one code smell instance, along with the percentage over the total number of files with self-admitted technical debt, for Long Method, Feature Envy, God Class, and all code smells combined, respectively.

JMeter, for example, has 200 files that contain self-admitted technical debt comments, and 143 of these files also contain at least one Long Method code smell (i.e., 71.5 percent). In addition, we can see that 20.5 percent of the files that have self-admitted technical debt are involved in Feature Envy code smells, and 48.5 percent of them are involved in God Class code smells. In summary, we see that 80.5 percent of the files that contain self-admitted technical debt comments are also involved in at least one of the three examined code smells. In general, we observe from Tables 7 and 8 that the overlap between self-admitted technical debt and code smells is higher than the ratio of files containing code smells. This indicates that there is some form of agreement between files that have code smells and files containing self-admitted technical debt.

We find that the code smell that overlaps the most with self-admitted technical debt is Long Method. Intuitively, this is expected, since Long Method is a common code smell and may have multiple instances per file, because it is computed at the method level. The overlap between files with self-admitted technical debt and Long Method ranged from 43.6 to 82 percent of all the files containing self-admitted technical debt comments, and considering all projects, the average overlap is 65 percent. In addition, 44.2 percent of the files with self-admitted technical debt comments are also involved in God Class code smells, and 20.7 percent in Feature Envy code smells. Taking all examined code smells in consideration we find that, on average, 69.7 percent of files containing self-admitted technical debt are also involved in at least one of the three examined code smells.

Our findings here shows that using code comments to identify technical debt is a complementary approach to using code smells to detect technical debt. Clearly, there is overlap, however, each approach also identifies unique instances of technical debt.

5 Related Work

Our work uses code comments to detect self-admitted technical debt using a Natural Language Processing technique. Therefore, we divide the related work into three sections, namely source code comments, technical debt, and NLP in software engineering.

5.1 Source Code Comments

A number of studies examined the co-evolution of source code comments and the rationale for changing code comments. For example, Fluri et al. [45] analyzed the co-evolution of source code and code comments, and found that 97 percent of the comment changes are consistent. Tan et al. [46] proposed a novel approach to identify inconsistencies between Javadoc comments and method signatures. Malik et al. [47] studied the likelihood of a comment to be updated and found that call dependencies, control statements, the age of the function containing the comment, and the number of co-changed dependent functions are the most important factors to predict comment updates.

Other works used code comments to understand developer tasks. For example. Storey et al. [30] analyzed how task annotations (e.g., TODO, FIXME) play a role in improving team articulation and communication. The work closest to ours is the work by Potdar and Shihab [14], where code comments were used to identify technical debt, called self-admitted technical debt.

Similar to some of the prior work, we also use source code comments to identify technical debt. However, our main focus is on the detection of different *types* of self-admitted technical debt. As we have shown, our approach yields different and better results in the detection of self-admitted technical debt.

5.2 Technical Debt

A number of studies has focused on the detection and management of technical debt. For example, Seaman et al. [2], Kruchten et al. [3] and Brown et al. [48] make several reflections about the term technical debt and how it has been used to communicate the issues that developers find in the code in a way that managers can understand.

Other work focused on the detection of technical debt. Zazworka et al. [13] conducted an experiment to compare the efficiency of automated tools in comparison with human elicitation regarding the detection of technical debt. They found that there is a small overlap between the two approaches, and thus it is better to combine them than replace one with the other. In addition, they concluded that automated tools are more efficient in finding defect debt, whereas developers can realize more abstract categories of technical debt.

In a follow up work, Zazworka et al. [49] conducted a study to measure the impact of technical debt on software quality. They focused on a particular kind of design debt, namely, God Classes. They found that God Classes are more likely to change, and therefore, have a higher impact on software quality. Fontana et al. [42] investigated design technical debt appearing in the form of code smells. They used metrics to find three different code smells, namely God Classes, Data Classes and Duplicated Code. They proposed an approach to classify which one of the different code smells should be addressed first, based on its risk. Ernst et al. [36] conducted a survey with 1,831 participants and found that architectural decisions are the most important source of technical debt.

Our work is different from the work that uses code smells to detect design technical debt, since we use code comments to detect technical debt. Moreover, our approach does not rely on code metrics and thresholds to identify technical debt and can be used to identify bad quality code symptoms other than bad smells.

More recently, Potdar and Shihab [14] extracted the comments of four projects and analyzed 101,762 comments to come up with 62 patterns that indicate self-admitted technical debt. Their findings show that 2.4-31 percent of the files in a project contain self-admitted technical debt. Bavota and Russo [50] replicated the study of self-admitted technical debt on a large set of Apache projects and confirmed the findings observed by Potdar and Shihab in their earlier work. Wehaibi et al. [51] examined the impact of self-admitted technical debt and found that self-admitted technical debt leads to more complex changes in the future. All three of the aforementioned studies used the comment patterns approach to detect self-admitted technical debt. Our earlier work [15] examined more than 33 thousands comments to classify the different types of self-admitted technical debt found in source code comments. Farias et al. [52] proposed a contextualized vocabulary model for identifying technical debt in comments using word classes and code tags in the process.

Our work also uses code comments to detect design technical debt. However, we use these code comments to train a maximum entropy classifier to automatically identify technical debt. Also, our focus is on *self-admitted* design and requirement technical debt.

5.3 NLP in Software Engineering

A number of studies leveraged NLP in software engineering, mainly for the traceability of requirements, program comprehension and software maintenance. For example, Lormans and van Deursen [53] used latent semantic indexing (LSI) to create traceable links between requirements and test cases and requirements to design implementations. Hayes et al. [54], [55] created a tool called RETRO that applies information retrieval techniques to trace and map requirements to designs. Yadla et al. [56] further enhanced the RETRO tool and linked requirements to issue reports. On the other hand, Runeson et al. [57] implemented a NLP-based tool to automatically identify duplicated issue reports, they found that 2/3 of the possible duplicates examined in their study can be found with their tool. Canfora and Cerulo [58] linked a change request with the corresponding set of source files using NLP techniques, and then, they evaluated the performance of the approach on four open source projects.

The prior work motivated us to use NLP techniques. However, our work is different from the aforementioned ones, since we apply NLP techniques on code comments to identify self-admitted technical debt, rather than use it for traceability and linking between different software artifacts.

6 Threats to Validity

Construct validity. considers the relationship between theory and observation, in case the measured variables do not measure the actual factors. When performing our study, we used well-commented Java projects. Since our approach heavily depends on code comments, our results and performance measures may be impacted by the quantity and quality of comments in a software project. Considering the intentional misrepresentation of measures, it is possible that even a well commented project does not contain self-admitted technical debt. Given the fact that the developers may opt to not express themselves in source code comments. In our study, we made sure that we choose case studies that are appropriately commented for our analysis. On the same point, using comments to determine some self-admitted technical debt may not be fully representative, since comments or code may not be updated consistently. However, previous work shows that changes in the source code are consistent with comment changes [14], [45]. In addition, it is possible that a variety of technical debt that is not self-admitted is present in the analyzed projects. However, since the focus of this paper is to improve the detection of the most common types of self-admitted technical debt, considering all technical debt is out of the scope of this paper.

Reliability validity. The training dataset used by us heavily relied on a manual analysis and classification of the code comments from the studied projects. Like any human activity, our manual classification is subject to personal bias. To reduce this bias, we took a statistically significant sample of our classified comments and asked a Master's student, who is not an author of the paper, to manually classify them. Then, we calculate the Kappa's level of agreement between the two classifications. The level of agreement obtained was +0.81, which according to Fleiss [33] is characterized as an excellent inter-rater agreement (values larger than +0.75 are considered excellent). Nevertheless, due to the irregular data distribution of our significant sample (which has many more comments without technical debt, than comments with the other classes of debt), we also measured Kappa's level of agreement for design and requirement self-admitted technical debt separately. The level of agreement obtained for design and requirement self-admitted technical debt was +0.75 and +0.84, respectively. Also, our approach depends on the correctness of the underlying tools we use. To mitigate this risk, we used tools that are commonly used by practitioners and by the research community, such as JDeodorant for the extraction of source code comments and for investigating the overlap with code smells (Section 4.4) and the Stanford Classifier for training and testing the max entropy classifier used in our approach.

External validity considers the generalization of our findings. All of our findings were derived from comments in open source projects. To minimize the threat to external validity, we chose open source projects from different domains. That said, our results may not generalize to other open source or commercial projects, projects written in different languages, projects from different domains and/or technology stacks. In particular, our results may not generalize to projects that have a low number or no comments or comments that are written in a language other than English.

7 Conclusion and Future Work

Technical debt is a term being used to express non-optimal solutions, such as hacks and workarounds, that are applied during the software development process. Although these non-optimal solutions can help achieve immediate pressing goals, most often they will have a negative impact on the project maintainability [49].

Our work focuses on the identification of self-admitted technical debt through the use of Natural Language Processing. We analyzed the comments of 10 open source projects namely Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby and SQuirrel SQL. These projects are considered well commented and they belong to different application domains. The comments of these projects were manually classified into specific types of technical debt such as design, requirement, defect, documentation and test debt. Next, we selected 61,664 comments from this dataset (i.e., those classified as design self-admitted technical debt, requirement self-admitted technical debt and without technical debt) to train the maximum entropy classifier, and then this classifier was used to identify design and requirement self-admitted technical debt automatically.

We first evaluated the performance of our approach by comparing the F1-measure of our approach with the F1-measure of two other baselines, i.e., the comment patterns baseline and the simple (random) baseline. We have shown that our approach outperforms the comment patterns baseline on average 2.3 and 6 times in the identification of design and requirement self-admitted technical debt, respectively. Moreover, our approach can identify requirement self-admitted technical debt, while the comment patterns baseline fails to detect this kind of debt in most of the examined projects. Furthermore, the performance of our approach surpasses the simple (random) baseline on average 7.6 and 19.1 times for design and requirement self-admitted technical debt, respectively.

Then, we explored the characteristics of the features (i.e., words) used to classify self-admitted technical debt. We find that the words used to express design and requirement self-admitted technical debt are different from each other. The three strongest indicators of design self-admitted technical debt are 'hack', 'workaround' and 'yuck!', whereas, 'todo', 'needed' and 'implementation' are the strongest indicators of requirement debt. In addition, we find that using only 5 and 23 percent of the comments in the training dataset still leads to an accuracy that is equivalent to 80 and 90 percent of the best performance, respectively. In fact, our results show that developers use a richer vocabulary to express design self-admitted technical debt and a training dataset of at least 3,900 comments (of which 195 comments are design self-admitted technical debt) is necessary to obtain a satisfactory classification. On the other hand, requirement self-admitted technical debt is expressed in a more uniform way, and with a training dataset of 2,600 comments (of which 52 are self-admitted technical debt) it is possible to classify with relatively high accuracy requirement self-admitted technical debt.

In the future, we believe that more analysis is needed to fine tune the use of the current training dataset in order to achieve maximum efficiency in the detection of self-admitted technical debt comments. For example, using subsets of our training dataset can be more suitable for some applications than using the whole dataset due to domain particularities. However, the results thus far are not to be neglected as our approach has the best F1-measure performance on every analyzed project. In addition, we plan to examine the applicability of our approach to more domains (than those we study in this paper) and software projects developed in different programming languages.

Another interesting research direction that we plan to investigate in the future is the use of other machine learning techniques, such as active learning to reduce the number of labeled data necessary to train the classifier. This technique, if proved successful, can further expand the horizon of projects that our approach can be applied to.

Moreover, to enable future research, we make the dataset created in this study publicly available.[1] We believe that it will be a good starting point for researchers interested in identifying technical debt through comments and even experimenting with different Natural Language Processing techniques. Lastly, we plan to use the findings of this study to build a tool that will support software engineers in the task of identifying and managing self-admitted technical debt.

1. https://github.com/maldonado/tse.satd.data

APPENDIX

Detailed Precision and Recall Values

In Section 3, we presented the F1-measure values for all projects when answering our research questions. In this appendix, we add the detailed precision and recall values that are used to compute the F1-measure values.

TABLE 9
Detailed Comparison of F1-Measure Between the NLP-Based, the Comment Patterns and the Random Baseline Approaches for Design Debt

Project	NLP-based			Comment Patterns			Random Baseline		
	Precision	Recall	F1 measure	Precision	Recall	F1 measure	Precision	Recall	F1 measure
Ant	0.554	0.484	0.517	0.608	0.147	0.237	0.023	0.5	0.044
ArgoUML	0.788	0.843	0.814	0.793	0.057	0.107	0.084	0.5	0.144
Columba	0.792	0.484	0.601	0.800	0.158	0.264	0.019	0.5	0.037
EMF	0.574	0.397	0.470	0.647	0.141	0.231	0.018	0.5	0.034
Hibernate	0.877	0.645	0.744	0.920	0.129	0.227	0.12	0.5	0.193
JEdit	0.779	0.378	0.509	0.857	0.214	0.342	0.019	0.5	0.037
JFreeChart	0.646	0.397	0.492	0.507	0.195	0.282	0.042	0.5	0.077
JMeter	0.808	0.668	0.731	0.813	0.110	0.194	0.039	0.5	0.072
JRuby	0.798	0.770	0.783	0.864	0.483	0.620	0.07	0.5	0.123
SQuirrel	0.544	0.536	0.540	0.700	0.100	0.175	0.029	0.5	0.055

TABLE 10
Detailed Comparison of F1-Measure Between the NLP-Based, the Comment Patterns and the Random Baseline Approaches for Requirement Debt

Project	NLP-based			Comment Patterns			Random Baseline		
	Precision	Recall	F1 measure	Precision	Recall	F1 measure	Precision	Recall	F1 measure
Ant	0.154	0.154	0.154	0.000	0.000	0.000	0.003	0.5	0.006
ArgoUML	0.663	0.540	0.595	0.000	0.000	0.000	0.043	0.5	0.079
Columba	0.755	0.860	0.804	0.375	0.069	0.117	0.007	0.5	0.013
EMF	0.800	0.250	0.381	0.000	0.000	0.000	0.004	0.5	0.007
Hibernate	0.610	0.391	0.476	0.000	0.000	0.000	0.022	0.5	0.041
JEdit	0.125	0.071	0.091	0.000	0.000	0.000	0.001	0.5	0.003
JFreeChart	0.220	0.600	0.321	0.102	0.266	0.148	0.003	0.5	0.007
JMeter	0.153	0.524	0.237	0.000	0.000	0.000	0.003	0.5	0.005
JRuby	0.686	0.318	0.435	0.573	0.318	0.409	0.022	0.5	0.043
SQuirrel	0.657	0.460	0.541	0.000	0.000	0.000	0.007	0.5	0.014

Detailed Precision and Recall Values When Using Different Classifiers

When discussing the results in Section 4.3, we only presented the F1-measure values. Here, we present the detailed precision and recall values that are used to compute the F1-measure values presented earlier.

TABLE 11
Comparison Between Different Classifiers for Design Debt

Project	Maximum Entropy			Naive Bayes			Binary		
	Precision	Recall	F1 measure	Precision	Recall	F1 measure	Precision	Recall	F1 measure
Ant	0.554	0.484	0.517	0.072	0.874	0.134	0.620	0.516	0.563
ArgoUML	0.788	0.843	0.814	0.358	0.985	0.525	0.790	0.858	0.822
Columba	0.792	0.484	0.601	0.181	0.786	0.294	0.840	0.500	0.627
EMF	0.574	0.397	0.470	0.057	0.872	0.106	0.633	0.397	0.488
Hibernate	0.877	0.645	0.744	0.288	0.890	0.435	0.895	0.670	0.767
JEdit	0.779	0.378	0.509	0.227	0.791	0.353	0.807	0.342	0.480
JFreeChart	0.646	0.397	0.492	0.140	0.560	0.224	0.658	0.397	0.495
JMeter	0.808	0.668	0.731	0.224	0.801	0.350	0.819	0.671	0.737
JRuby	0.798	0.770	0.783	0.275	0.971	0.429	0.815	0.808	0.811
SQuirrel	0.544	0.536	0.540	0.133	0.947	0.233	0.567	0.550	0.558
Average	0.716	0.5602	0.6201	0.1955	0.8477	0.3083	0.7444	0.5709	0.6348

TABLE 12
Comparison Between Different Classifiers for Requirement Debt

Project	Maximum Entropy			Naive Bayes			Binary		
	Precision	Recall	F1 measure	Precision	Recall	F1 measure	Precision	Recall	F1 measure
Ant	0.154	0.154	0.154	0.007	0.769	0.013	0.188	0.231	0.207
ArgoUML	0.663	0.540	0.595	0.119	0.808	0.207	0.659	0.569	0.611
Columba	0.755	0.860	0.804	0.030	0.930	0.057	0.755	0.860	0.804
EMF	0.800	0.250	0.381	0.009	1.000	0.018	0.800	0.250	0.381
Hibernate	0.610	0.391	0.476	0.041	0.781	0.078	0.615	0.375	0.466
JEdit	0.125	0.071	0.091	0.011	0.857	0.022	0.143	0.071	0.095
JFreeChart	0.220	0.600	0.321	0.009	0.800	0.018	0.179	0.467	0.259
JMeter	0.153	0.524	0.237	0.011	0.952	0.022	0.180	0.524	0.268
JRuby	0.686	0.318	0.435	0.058	0.836	0.109	0.679	0.327	0.442
SQuirrel	0.657	0.460	0.541	0.018	0.900	0.036	0.455	0.500	0.476
Average	0.4823	0.4168	0.4035	0.0313	0.8633	0.058	0.4653	0.4174	0.4009

REFERENCES

[1] W. Cunningham, "The WyCash portfolio management system," in *Proc. Object-Oriented Program. Syst. Languages Appl.*, 1992, pp. 29–30.

[2] C. Seaman and Y. Guo, "Chapter 2—measuring and monitoring technical debt," in *Advances in Computers*, vol. 82, M. V. Zelkowitz, Ed. Amsterdam, The Netherlands: Elsevier, 2011, pp. 25–46.

[3] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi, "Technical debt: Towards a crisper definition report on the 4th International Workshop on Managing Technical Debt," *ACM SIGSOFT Softw. Eng. Notes*, vol. 38, pp. 51–54, 2013.

[4] E. Lim, N. Taksande, and C. Seaman, "A balancing act: What software practitioners have to say about technical debt," *IEEE Softw.*, vol. 29, no. 6, pp. 22–27, Nov./Dec. 2012.

[5] M. Fowler, "Technical debt quadrant," (2009). [Online]. Available: http://martinfowler.com/bliki/TechnicalDebtQuadrant.html, Accessed on: Jun. 09, 2016.

[6] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *Proc. Joint Work. IEEE/IFIP Conf. Softw. Archit. Eur. Conf. Softw. Archit.*, 2012, pp. 91–100.

[7] N. Alves, T. Mendes, M. G. de Mendonca, R. Spinola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Inf. Softw. Technol.*, vol. 70, pp. 100–121, 2016.

[8] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 488–498.

[9] N. Alves, L. Ribeiro, V. Caires, T. Mendes, and R. Spinola, "Towards an ontology of terms on technical debt," in *Proc. 6th Int. Workshop Manag. Tech. Debt*, 2014, pp. 1–7.

[10] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM J. Res. Develop.*, vol. 56, pp. 1–13, 2012.

[11] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. 20th IEEE Int. Conf. Softw. Maintenance*, 2004, pp. 350–359.

[12] R. Marinescu, G. Ganea, and I. Verebi, "Incode: Continuous quality assessment and improvement," in *Proc. 14th Eur. Conf. Softw. Maintenance Reengineering*, 2010, pp. 274–275.

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

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

[15] E. D. S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *Proc. 7th Int. Workshop Manag. Tech. Debt*, 2015, pp. 9–15.

[16] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *J. Syst. Softw.*, vol. 84, no. 10, pp. 1757–1782, Oct. 2011.

[17] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Trans. Softw. Eng.*, vol. 41, no. 11, pp. 1055–1090, Nov. 2015.

[18] J. Graf, "Speeding up context-, object- and field-sensitive SDG generation," in *Proc. 10th IEEE Work. Conf. Source Code Anal. Manipulation*, 2010, pp. 105–114.

[19] K. Ali and O. Lhoták, "Application-only call graph construction," in *Proc. 26th Eur. Conf. Object-Oriented Program.*, 2012, pp. 688–712.

[20] P. Oliveira, M. Valente, and F. Paim Lima, "Extracting relative thresholds for source code metrics," in *Proc IEEE Conf. Softw. Maintenance Reengineering Reverse Eng.*, 2014, pp. 254–263.

[21] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," in *Proc. 6th Int. Workshop Emerging Trends Softw. Metrics*, 2015, pp. 44–53.

[22] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Softw. Eng.*, vol. 21, pp. 1–49, 2015.

[23] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Antipattern and code smell false positives: Preliminary conceptualization and classification," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineering*, 2016, pp. 609–613.

[24] C. Vassallo, F. Zampetti, D. Romano, M. Beller, A. Panichella, M. D. Penta, and A. Zaidman, "Continuous delivery practices in a large financial organization," in *Proc. 32nd Int. Conf. Softw. Maintenance Evol.*, 2016, pp. 519–528.

[25] C. Manning and D. Klein, "Optimization, maxent models, and conditional estimation without magic," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics Human Language Tech.*, 2003, pp. 8–8.

[26] E. Maldonado, E. Shihab, and N. Tsantalis, "Replication package for using natural language processing to automatically detect self-admitted technical debt," 2016. [Online]. Available: https://github.com/maldonado/tse_satd_data/

[27] D. A. Wheeler, "SLOC count users guide," 2004. [Online]. Available: http://www.dwheeler.com/sloccount/sloccount.html

[28] "OpenHub homepage," (2016). [Online]. Available: https://www.openhub.net/, Accessed on: Dec. 12, 2014.

[29] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *Proc. 12th Eur. Conf. Softw. Maintenance Reengineering*, 2008, pp. 329–331.

[30] M. Storey, J. Ryall, R. Bull, D. Myers, and J. Singer, "Todo or to bug," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 251–260.

[31] J. Cohen, "A coefficient of agreement for nominal scales," *Educ. Psychological Meas.*, vol. 20, pp. 37–46, 1960.

[32] J. L. Fleiss and J. Cohen, "The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability," *Educ. Psychological Meas.*, vol. 33, pp. 613–619, 1973.

[33] J. Fleiss, "The measurement of interrater agreement," in *Statistical Methods Rates Proportions*. NewYork, NY, USA: Wiley, 1981, pp. 212–236.

[34] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction Inf. Retrieval*. Cambridge, U.K.: Cambridge University Press, 2008.

[35] A. Bessey, et al., "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, pp. 66–75, 2010.

[36] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proc. 10th Joint Meet. Found. Softw. Eng.*, 2015, pp. 50–60.

[37] C. Sadowski, J. V. Gogh, C. Jaspan, E. Soderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, 2015, pp. 598–608.

[38] D. Berry, R. Gacitua, P. Sawyer, and S. F. Tjong, *The Case for Dumb Requirements Engineering Tools*. Berlin, Germany: Springer, 2012.

[39] G. Forman and I. Cohen, "Learning from little: Comparison of classifiers given little training," in *Proc. 8th Eur. Conf. Principles Practice Knowl. Discovery Databases*, 2004, pp. 161–172.

[40] A. Y. Ng and M. I. Jordan, "On discriminative versus generative classifiers: A comparison of logistic regression and naive Bayes." in *Proc. Neural Inf. Process. Syst.*, 2001, pp. 841–848.

[41] M. Banko and E. Brill, "Scaling to very very large corpora for natural language disambiguation," in *Proc. 39th Annu. Meet. Assoc. Comput. Linguistics*, 2001, pp. 26–33.

[42] F. Fontana, V. Ferme, and S. Spinelli, "Investigating the impact of code smells debt on quality code evaluation," in *Proc. 3rd Int. Workshop Manag. Tech. Debt*, 2012, pp. 15–22.

[43] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *Proc. IEEE Int Conf. Softw. Maintenance*, Sept. 2010, pp. 1–10.

[44] D. I. Sjoberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013.

[45] B. Fluri, M. Wursch, and H. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *Proc. 14th Work. Conf. Reverse Eng.*, 2007, pp. 70–79.

[46] S. H. Tan, D. Marinov, L. Tan, and G. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *Proc. IEEE 5th Int. Conf. Softw. Testing Verification Validation*, 2012, pp. 260–269.

[47] H. Malik, I. Chowdhury, T. Hsiao-Ming , Z. M. Jiang, and A. Hassan, "Understanding the rationale for updating a function comment," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2008, pp. 167–176.

[48] N. Brown, et al., "Managing technical debt in software-reliant systems," in *Proc. FSE/SDP Workshop Future Softw. Eng. Res.*, 2010, pp. 47–52.

[49] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proc. 2nd Int. Workshop Manag. Tech. Debt*, 2011, pp. 17–23.

[50] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proc. 13th Int. Workshop Mining Softw. Repositories*, 2016, pp. 315–326.

[51] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineering*, 2016, pp. 179–188.

[52] M. A. D. F. Farias, M. G. D. M. Neto, A. B. D. Silva, and R. O. Spinola, "A contextualized vocabulary model for identifying technical debt on code comments," in *Proc. 7th Int. Workshop Manag. Tech. Debt*, 2015, pp. 25–32.

[53] M. Lormans and A. Van Deursen , "Can LSI help reconstructing requirements traceability in design and test?" in *Proc. 10th Eur. Conf. Softw. Maintenance Reengineering*, 2006, pp. 47–56.

[54] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Improving after-the-fact tracing and mapping: Supporting software quality predictions," *IEEE Softw.*, vol. 22, no. 6, pp. 30–37, Nov./Dec. 2005.

[55] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods," *IEEE Trans. Softw. Eng.*, vol. 32, no. 1, pp. 4–19, Jan. 2006.

[56] S. Yadla, J. H. Hayes, and A. Dekhtyar, "Tracing requirements to defect reports: An application of information retrieval techniques," *Innovations Syst. Softw. Eng.*, vol. 1, pp. 116–124, 2005.

[57] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 499–510.

[58] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *Proc. 11th IEEE Int. Symp. Softw. Metrics*, 2005, pp. 21–29.

Everton da Silva Maldonado received the BS degree in information systems from the University Impacta of Technology, Brazil, in 2009 and the MS degree in software engineering from Concordia University, Canada, in 2016. His main research interests include mining software repositories, software quality assurance, technical debt, and software maintenance.

Emad Shihab received the PhD degree from Queens University. He is an assistant professor in the Department of Computer Science and Software Engineering, Concordia University. His research interests include software quality assurance, mining software repositories, technical debt, mobile applications, and software architecture. He worked as a software research intern with Research In Motion, Waterloo, Ontario and Microsoft Research, Redmond, Washington. He is a member of the IEEE and the ACM. More information can be found at http://das.encs.concordia.ca

Nikolaos Tsantalis received the PhD degree in computer science from the University of Macedonia, Thessaloniki, Greece, in 2010. He is an assistant professor in the Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, and holds a Concordia University Research Chair in Web Software Technologies. From January 2011 until May 2012, he was a postdoctoral fellow in the Department of Computing Science, University of Alberta, Edmonton, Canada. His research interests include software maintenance, empirical software engineering, refactoring recommendation systems, and software quality assurance. In 2016, he has been awarded with an ACM SIGSOFT Distinguished Paper Award and an ACM SIGSOFT Distinguished Artifact Award at FSE. He serves regularly as a program committee member of international conferences in the field of software engineering, such as ASE, ICSME, SANER, ICPC, and SCAM. He is a member of the IEEE and the ACM, and holds a license from the Association of Professional Engineers of Ontario.