**Authors’ response**

Editor Comments  
  
Editor  
Comments to the Author:  
The reviewers all find novelty in the work of looking for technical debt in comments. However, there are several areas in which the manuscript requires improvement. In particular, the authors should:  
  
\* provide arguments or justification of why the studies focus on comments and do not validate the technical debt with the source code and whether it exhibits characteristics of technical debt. This point is related to the reviewers comments on choices not to compare to ways to assess technical debt based on source code.  
  
\* clarify how technical debt and requirements debt relate. The authors should carefully consider what information to fold in from the previous workshop paper to enable readers to have a self-contained means of understanding this manuscript.  
  
\* several reviewers commented on finding choices made for the random classifier (such as following the distribution of technical debt on the projects): these require clarification.  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
Reviewers' Comments  
  
Please note that some reviewers may have included additional comments in a separate file. If a review contains the note "see the attached file" under Section III A - Public Comments, you will need to log on to ScholarOne Manuscripts  to view the file. After logging in, select the Author Center, click on the "Manuscripts with Decisions" queue and then clicking on the "view decision letter" link for this manuscript. You must scroll down to the very bottom of the letter to see the file(s), if any.  This will open the file that the reviewer(s) or the Associate Editor included for you along with their review.

**Reviewer: 1**  
  
Public Comments (these will be made available to the author)  
The paper presents an approach to detect technical debt automatically from source code comments based on NLP. The approach is used on a number of open source systems to train the NLP classifier and provide validation. The authors also discuss the prominent words used to indicate design and requirements technical debt and argue that their approach only needs a small training set.  
The paper is easy to read and understand, written in a concise and clear manner. It tackles an important problem and reports on a well-design empirical study that provides sound evidence. However there are several points that need to be revised regarding the **motivation for the work**, the **design of the approach** and the **empirical study**. Detailed comments per section follow.

Introduction   
• (R1-1) Technical debt is not always incurred consciously. Martin Fowler was first to point this out.

Thank you for the comment. Indeed, we agree with the reviewer that technical debt can be incurred consciously and unconsciously. Clearly, in this work we focus on the self-admitted technical debt, which would fall under the technical debt that is consciously (or deliberately as Fowler calls it) incurred. We clarified this point in the paper and added the following text to the manuscript in the Introduction, third paragraph:

“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.”

As part of this change, we also added the following citations:

[5]  “M. Fowler. Technical debt quadrant.”   
http://martinfowler.com/bliki/TechnicalDebtQuadrant.html, accessed: 2016-06-09.

[6]  R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, “In search of a metric for managing architectural technical debt,” in *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, 2012, pp. 91–100.

• (R1-2) Static analysis does help in detecting debt and is the more popular approach but there are also other techniques proposed that you seem to completely ignore. I know you want to compare your approach with techniques that focus on source code but you cannot ignore other ways of detecting debt, like architecture reviews.

Thank you for the comment. We mainly focused on the source code-related technical debt since our approach is related to source code comments. However, as the reviewer points out it is a good idea to also mention other ways that technical debt has been measured in the past. Therefore, we modified the manuscript to add the following text in Introduction, paragraph 4:

“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] ”

As part of this change, we also added the following citations:

* [7]  N. Alves, T. Mendes, M. G. de Mendona, R. Spinola, F. Shull, and C. Seaman, “Identification and management of technical debt: A systematic mapping study,” *Information and Software Technology*, vol. 70, pp. 100–121, 2016.
* [8]  L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, “Identifying and quantifying architectural debt,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 488–498.

• [NIKOS](R1-3) You argue that detection based on source comments is better than source code, but you don’t justify it well. First, I don’t buy your third argument: why exactly is it more reliable to base on comments? Yes, developers write comments themselves but why does it make this approach more reliable? Also the first argument is not clear. So what if source code techniques require an AST? Such tools produce the end result in seconds so what is the problem exactly? In fact I am not at all convinced that detecting based on source comments is better at all – it is at best a complementary way.

Thank you for this comment. We have removed the third argument and emphasized the complementary role of detecting TD based on source comments. In addition, we supported the other two arguments with references from the related literature. The updated text is as follows:

“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 well 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 is definitely more relevant to them.”

Approach  
• (R1-4) The approach looks in general sound but the manual classification is a major issue. First, spending 185 hours to perform the manual classification is major effort. Anyone that wants to reuse your approach will have to retrain the classifier and perform this manual classification from scratch for a different domain, language, technology stack. This is a major hindrance towards the applicability of your approach.

Thank you for the comment. Indeed, the manual effort put into the approach is significant, however, the main purpose of the manuscript is to put forward an approach that can be trained using our manually classified dataset, so that such manual effort can be reduced in the future. As we have shown in our RQ3 results, our NLP classifier performs well, even when tested on a project that has never been trained on in the past. We also make our dataset publicly available so that others easily build on our dataset or extend it.

As for the domain issue, we specifically chose our case studies to cover different domains. That said, we do not claim that our approach will generalize to all languages, domains, etc; achieving such generalizations is out of the scope of our paper and requires a different study specifically designed for such a purpose. To address this comment and clarify our intention here, we modified/added the following text to Sections 6 (Threats to Validity) and 7 (Conclusions and Future Work):

Section 6:

“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 that are written on a different idiom than English.”

Section 7:

“In addition, we plan to examine the applicability of our approach to more domains (than what we study in this paper) and software projects developed in different programming languages.”

• (R1-5) Second, inter-rate agreement measured according to Cohen’s Kappa is 0.81 which is good but not impressive. This should be added to your threats to validity

Thank you for the comment. We made sure to discuss the inter-rater agreement in the threats to validity of the original submission. However, we believe that the reviewer’s comment is related to the value achieved. After the reviewer’s comment, we investigated the characterization of different Cohen’s Kappa values. The literature shows that a Cohen’s Kappa value of < 0.4 as being poor, 0.40 - 0.75 as being fair to good, and > 0.75 as excellent. To address this comment, we modified our original text in Section 6, first paragraph:

“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 [46] is characterized as an excellent inter-rater agreement (values larger than 0.75 are considered excellent).”

As part of this change, we also added the following citation:

[46] J. Fleiss, “The measurement of interrater agreement,” *Statistical methods for rates and proportions.*, pp. 212–236, 1981.

• (R1-6) The selected OSS projects are at most a couple hundred thousands SLOC. Does this pose a threat to the external validity? Would your approach work on larger systems? Also please provide details on the application domains and explain in your threats to validity whether you can generalize to other application domains.

Thank you for the comment. As shown in Table 1, our projects range in size between ~81K - ~228K SLOC. This is comparable to other technical debt-related studies. For example, in [36], the authors study 2 projects that are 35K and 45K LOC, another study [37] used 12 subject systems that range in size between ~25K - ~81K. Therefore, we believe that our studied systems are in line with prior work. As for whether our approach would work on larger systems, we do not see any reason why the approach would not work on larger systems. The real question to consider however, is the quantity and quality of comments in these system, rather than their size. We mention that the quantity and quality of comments is a key component that may affect our approach. The following text in Section 6, paragraph 2:

“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.”

As for the last part of the comment, we added the domain description for each application in Section 2.1, first paragraph:

“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 test functional behaviour 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.”

Regarding the generalizability of our approach, we believe that this is related to comment R1-4 and has been addressed in that comment.

[36] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, “Investigating the impact of design debt on software quality,” in *Proceedings of the 2nd International Workshop on Managing Technical Debt*, 2011, pp. 17–23.

[37] F. Fontana, V. Ferme, and S. Spinelli, “Investigating the impact of code smells debt on quality code evaluation,” in *Proceedings of the 3rd International Workshop on Managing Technical Debt*, 2012, pp. 15–22.

Case Study results  
• (R1-7) When formulating RQ1, do you consider that identifying technical debt is the same as predicting it? Also what exactly does effectiveness mean in this case?

Thank you for the comment. Indeed, we agree with the reviewer that identifying and predicting technical debt are distinct things, and should therefore be addressed as such. While predicting technical debt is related to time (using older debt to predict future debt), identifying technical debt is related to distinguish between the different types of debt. Thus, our approach is about identifying self-admitted technical debt. To address the reviewer comment we modified the wording of RQ1 to make it clearer as follows:

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

As for what we mean by effectiveness, in this case we would like our approach to identify the SATD comments with accuracy that is better than the state-of-the-art today (which is using comment patterns). To address this comment we added the following clarification in section 3 paragraph 4

“Therefore, we want to determine if NLP tools such as, the maximum entropy classifier, can help us surpass these limitations and outperform the accuracy of the current state-of-the-art.”

• (R1-8) I find the improvement over the comment patterns approach convincing (for design debt), but the comparison with the random approach (for requirements debt) is really not impressive. The F1-measure looks fine without making this comparison. This 18x improvement sounds like an oversell considering you only compare to the random classifier.

Thank you for the comment. Due the fact that self-admitted technical debt entries are very few in the studied dataset, the random classifier F1-measure serves as a **lower bound baseline**. Our intention when comparing our approach to the baseline is to make sure that we are at least better than this lower bound. That said, we do see the reviewer’s point about the 18X improvement being an oversell, since it is comparing to a lower bound. Therefore, we modified the manuscript removing the reference to the 18x improvement in the conclusion box for RQ1 as follow:

“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 2.3x over the state-of-the-art approach) and an average F1-measure of 0.403 when identifying requirement debt (an average improvement of 6x over the state-of-the-art approach).”

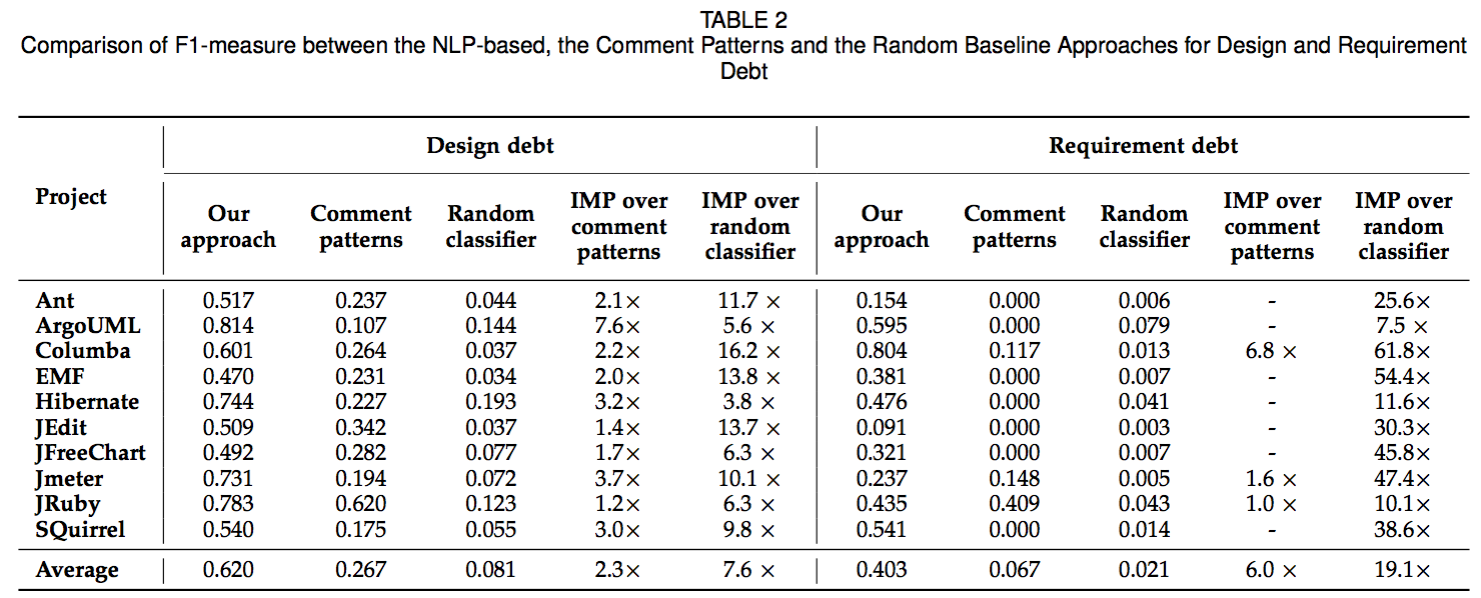
• (R1-9) Also, why wasn’t the comment patterns approach not able to detect any requirements debt?

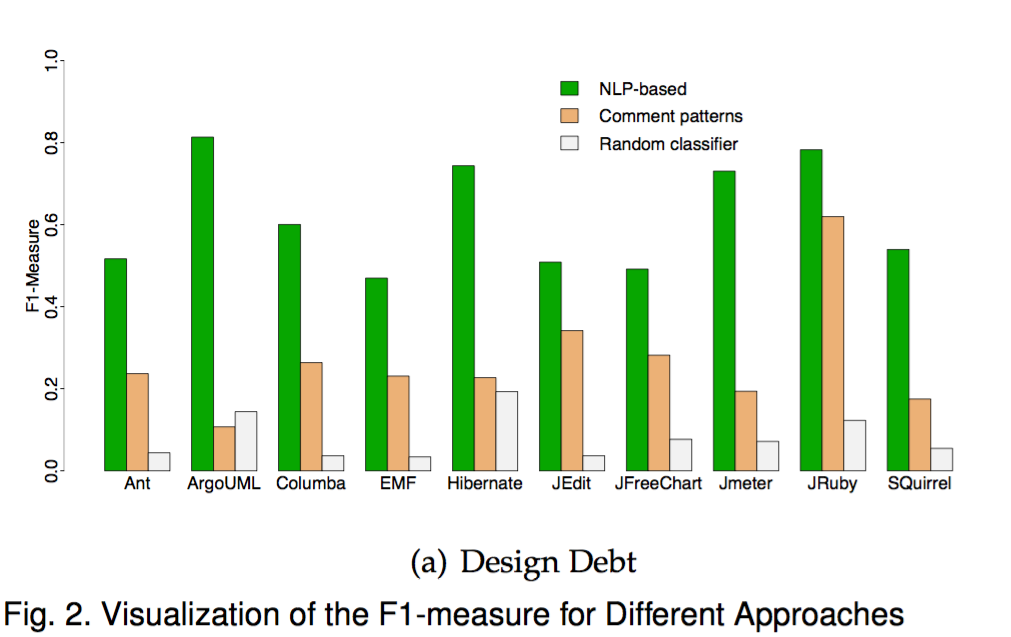
Thank you for the comment. To give a proper answer to this comment we revisited the experiment that was made to measure the performance of the comment patterns approach while identifying requirement technical debt. We found an issue in our scripts related to the case of the words in the comment patterns, which impacted the detection of the comment patterns approach. We resolved the issue and double checked all of our scripts to make sure the results are correct. After doing so, we find that the comment patterns approach is able to detect some requirement debt in 3 projects. Although this is still a poor performance, it is better than being unable to detect any requirement debt.

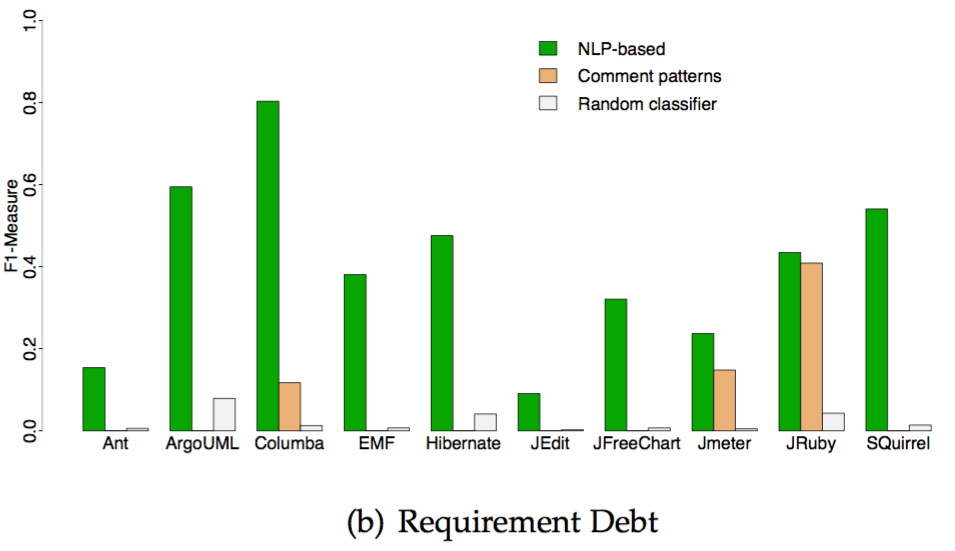
To address this comment, we modified our original text in Section 3, paragraph 13:

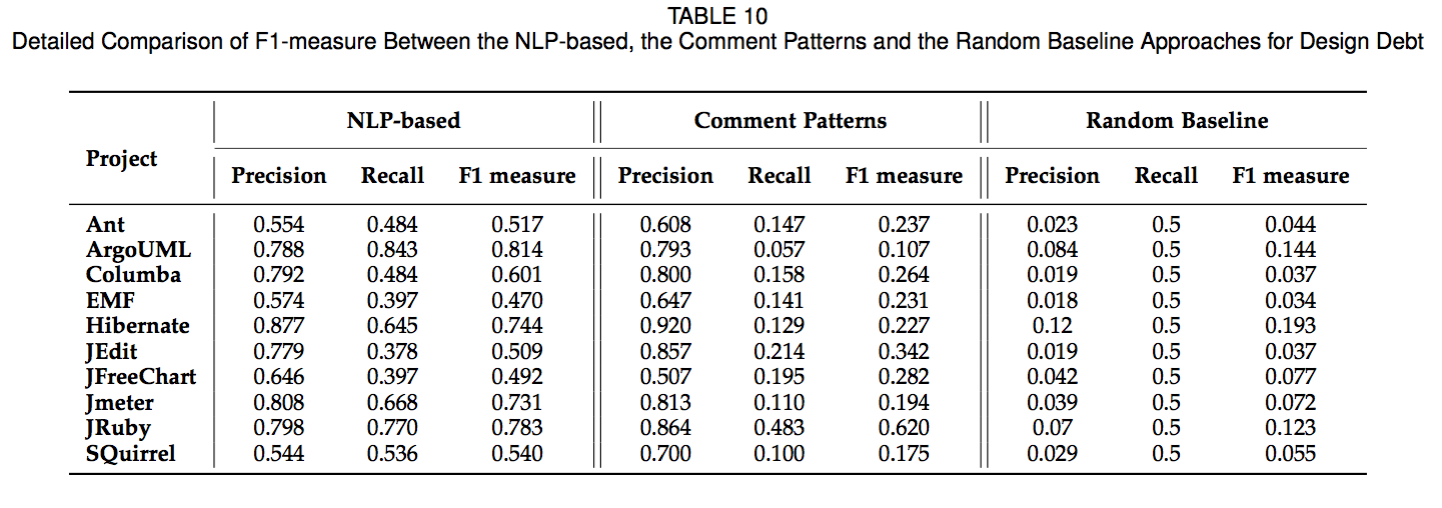
“Similarly, the last five columns of Table 2 show the F1-measure performance of the three approaches, and the improvement achieved by our approach over the two other approaches. The comment patterns approach was able to identify requirement self-admitted technical debt in only 3 of the 10 analyzed projects. 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.”

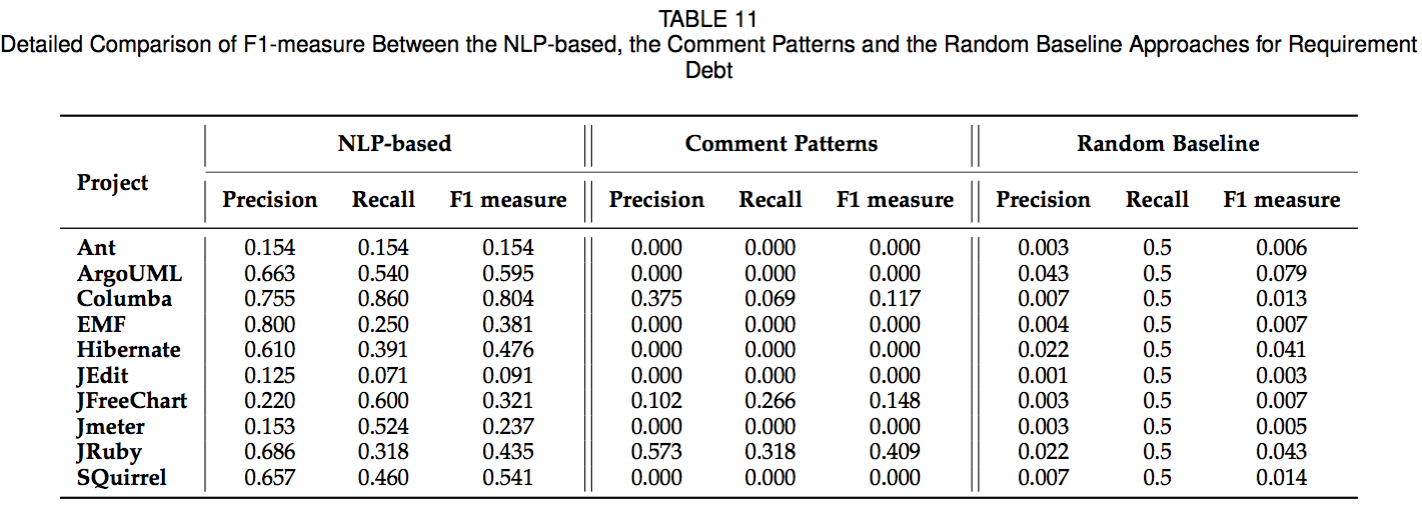
To address this comment and reflect the update to the comment pattern results, we updated the following tables and figures: section 3 Table 2, section 3 Figure 2a, section 3 Figure 2b, appendix Tables 7 and 8.











• (R1-10) One thing you do not explain is what you consider as design debt vs. requirement debt. This is a threat to your construct validity.

Thank you for the comment. We explained what we consider design and requirement debt in our previous work, and this information was provided in the original text through a citation [15]. However, we agree with the reviewer that the paper needs to stand on its own, hence, we added the following text from our previous work to better explain and give examples of what we consider as in section 2.4 paragraphs 3 – 6.

“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 re-implementing *existing* code to make it faster, more secure, more stable and so on. 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 probably 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]

The citation mentioned and that was used to add this text is the following:

[15] E. d. S. Maldonado and E. Shihab, “Detecting and quantifying different types of self-admitted technical debt,” in *Proceedings of the 7th International Workshop on Managing Technical Debt*, 2015, pp. 9–15.

• (R1-11) If I look at the keywords from RQ2 I am really not sure how you classified requirements debt. The fact that you have the same keyword appearing in both design and requirements debt indicates you may not have a clear distinction. To make matters worse, what you seem to hint is that requirements debt concerns requirements not yet implemented in code. This is in contrast to the orthodox perception on technical debt (see P. Kruchten et al. Technical Debt: From Metaphor to Theory and Practice)

Thank you for the comment. We believe that the reviewer comment is related to the missing explanation about what we considered as design and requirement self-admitted technical debt. We hope that this issue is addressed through the previous comments, R1-10.

In P. Kruchten et al. Technical Debt: From Metaphor to Theory and Practice, the authors explain how the technical debt metaphor is getting traction over the years and how multiple authors have been using the metaphor to communicate “not quite right code”. The authors express their concern about how the use (or abuse) of the metaphor could spread it too thin making the metaphor lose its communication power. More recently, N. Alves et al. in their paper, “Identification and management of technical debt: A systematic mapping study” defined requirement debt as:

“Requirements debt: Refers to trade-offs made with respect to what requirements the development team needs to implement or how to implement them. Some examples of this type of debt are: requirements that are only partially implemented, requirements that are implemented but not for all cases, requirements that are implemented but in a way that doesn't fully satisfy all the non-functional requirements (e.g. security, performance, etc.)”

We believe that our definition of requirement debt is not in disagreement with the literature. That said, we agree with the reviewer that our original text may lead the reader to conclude that requirement debt is related to requirements not yet implemented in code instead of *partially* implemented requirements. To address this problem, we provide some indicative comments using the top-ranked features shown in Table 3 that constitute design and requirement debt in the following text on RQ2, paragraph 7.

“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]”

**Please note in the aforementioned examples how TODO is used to express both design and requirement debt when used in combination with other words, or how the word need/needed changes the meaning of the comment when used with and without “?”.**

We hope that the provided explanation and the modified text address this issue, however, if it has not, we are happy to incorporate any other changes the reviewers suggest.

P. Kruchten, R. L. Nord, and I. Ozkaya, “Technical debt: From metaphor to theory and practice.” *Ieee software*, vol. 29, 2012.

N. Alves, T. Mendes, M. G. de Mendonça, R. Spinola, F. Shull, and C. Seaman, “Identification and management of technical debt: A systematic mapping study,” *Information and Software Technology*, vol. 70, pp. 100–121, 2016.

Discussion  
• (R1-12) The similarity of terms in requirements and design debt cannot be intuitively confirmed by looking at the top-ten terms from RQ2. There it looks like some terms (e.g. convention, configurable, apparently, fudging) are less similar than those for design debt. Can you explain that?

Thank you for the comment. In RQ2 we display the top-ten terms that were used to identify design and requirement debt. This means that these terms have the highest weight between all the other terms that were used during the classification process. The weight is given through the number of occurrences that a feature has in the training data. Moreover, the classification process is not based only on the top-ten terms, but on a combination of terms (as many as the classifier can match) to determine the class of the comment. Therefore, the top-ten words do not necessarily need to be similar with each other or have a semantic overlap.

However, to provide more insight we highlighted in Table 3 the words that appear consistently in all top-10 lists extracted from each one of the training data sets (i.e., first two words for design debt, and first 5 words for requirement debt). We consider these words as more universal features compared to the others. Although the non-highlighted features have still large weights, they can be considered as more project specific. We also added the following text in the paper:

“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 features compared to the others.”

Threats to validity  
•  (R1-13) It seems you have confused internal validity. It concerns causality which you do not study in your paper. The threats you mention, like the bias during manual classification is a threat to construct validity.

Thank you for the comment. We revisited the threats to validity sections and marked the internal and construct validity in the manuscript as suggested by the reviewer.

• (R1-14) Please extend your discussion with threats to construct validity and reliability

Thank you for the comment. We added the following text in the manuscript on section 6 paragraphs 3 and 4 to address this comment.

“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 chose case studies that were well commented for our analysis.

Lastly, 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 and the Stanford Classifier.”

Small Details  
•       “this is a dirty hack it’s better do to something” -> to do  
•       “”conjucture”

Thank you for the comment. We fixed the aforementioned typos.

**Reviewer: 2**  
Public Comments (these will be made available to the author)  
This paper presents a NLP approach to detecting 'self-admitted technical debt', that is, comments in the code that confess to (presumably) the following code being debt-laden. The authors previously approached this with a word-bag model which identified TD using common terms (like "hack"). In this paper they extend the approach using several NLP techniques to create a classifier from their large gold-standard of manually labeled TD items. They show significant improvement over the previous approach and a random approach to classification. They also present some results showing sensitivity of the classifier to training set size, which is useful for industry applications.  
  
• (R2-1) The paper is well written, aside from frequent reference to their earlier work [10].

Thank you for the comment. Indeed, there are multiple citations to this particular work. The main reason is that this study was the first one to explore technical debt found in source code comments, and we use the approach suggested in this paper as our upper boundary baseline. That said, we agree with the reviewer that the frequent reference to this previous work should be decreased. To address this comment we removed the reference to it on several parts such as Introduction paragraphs 8 and 9, subsection 2.3 paragraph 2, RQ1 paragraphs 1 and 6 and RQ2 paragraph 8.

------------------------------------------------------------------------------------------------------------------

We should reduce the frequency of citations to [10].

------------------------------------------------------------------------------------------------------------------

• (R2-2) The idea of detecting TD from code comments using NLP tools is novel and empirically demonstrated in the paper; a tool that is publicly available (for eg. in SonarQube) would be interesting to test with developers. However, there is no connection made between the authors' construct of self-admitted TD, with other notions of TD. Therefore, the study is really just a labeling exercise using categories defined in another paper, then the use of an off-the-shelf NLP tool.

Thank you for the comment. During this work we invested a lot of effort to make a solid contribution that would serve to the research community as well to practitioners. For instance, we added 5 more projects to our manually classified dataset. Creating such dataset is a laborious and complex task that fulfils a space that, for the best of our knowledge, has no precedents. This dataset is publicly available, and can enable future research in an area that is almost unexplored to the moment (i.e., finding technical debt through source code comments).

That said, we think that the reviewer had the impression that self-admitted technical debt has no connection with “actual” technical debt that can be detected by static analysis tools, which is not completely right. However, there is no text in the original manuscript to show otherwise. To address this comment we compare the overlap between technical debt files identified by static analysis tool and our approach. The details and the text added into the manuscript can be found in the next comment made by the reviewer. (R2-3)

------------------------------------------------------------------------------------------------------------------

We should clarify that the journal paper adds new datasets and also performs an empirical study to show the effectiveness of NLP in detecting SATD. We also share our data, which enables future studies on SATD.

------------------------------------------------------------------------------------------------------------------

• (R2-3) To demonstrate a useful contribution, the work should try to validate the labeling (and subsequent classifier) against either other static analysis tools, or with working developers. That way readers will know whether the classifier is actually detecting TD or not.

Thank you for the comment. Indeed, comparing our approach to detect self-admitted technical with other static analysis tools is a very good idea. To address this comment we compared the technical debt files that were detected by our approach and the files detected by a static analysis tool. More specifically, we used JDeodorant to detect three well know code smells (i.e., long method, feature envy and god class) that are commonly considered as technical debt. We present and discuss the results in subsection 4.4 under the Discussion section.

We also tried a metric-based code smell detector “<https://github.com/diegocedrim/code-smells-detector>” which is one of the very few available tools that supports the detection of Feature Envy, Long Method, and God/Blob Class and can be executed from command-line without depending on an IDE, and thus was ideal for our experiment.

However, we found out that it was flagging too many files as problematic, especially for Feature Envy code smell.

After a careful inspection of the tool's code, we realized that the implementation for Feature Envy is not correct, because it flags a method as feature envy if the total number of dependencies to other classes is more than the dependencies to the class the method currently belong to.

The correct implementation would need to count the number of dependencies individually for each external class, and flag the method as Feature Envy if the number of dependencies to one of the external classes is more than the number of internal dependencies. Due to this implementation error, we considered that the tool is not reliable enough, and thus we used JDeodorant.

“**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 the most established approach to identify technical debt in the source code. 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.

In this subsection, 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.

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. These code smells have been extensively investigated in the literature, and are considered to occur frequently [40], [41].

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 9 provides details about each of the projects used in our study. The columns of Table 9 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 combined code smells, 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%). In addition, we can see that 20.5% of the files that have self-admitted technical debt are involved in Feature Envy code smells, and 48.5% of them are involved in God Class code smells. In summary, we see that 80.5% of the files that contain self-admitted technical debt comments are also involved in at least one of the three examined code smells.

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 method level.

The overlap between files with self-admitted technical debt and Long Method ranged from 43.6% to 82% of all the files containing self-admitted technical debt comments, and considering all projects, the average overlap is 65%. In addition, 44.2% of the files with self-admitted technical debt comments are also involved in God Class code smells, and 20.7% in Feature Envy code smells. Taking all examined code smells in consideration we find that, on average, 69.7% of files containing self-admitted technical debt are also involved in at least one of the three examined code smells”



As part of this change, we also added the following citations:

[23]  F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, “Antipattern and code smell false positives: Preliminary conceptualization and classification,” in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, 2016, pp. 609– 613.

[40]  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 *Software Maintenance (ICSM), 2010 IEEE International Conference on*, Sept 2010, pp. 1–10.

[41]  D. I. Sjoberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dyba, “Quantifying the effect of code smells on maintenance effort,” *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144– 1156, 2013.

------------------------------------------------------------------------------------------------------------------

Thank you for the suggestion, comparing to a tool such as SonarQube is a good idea (and we should add that). We expected (and as we will see) that these two approaches are complementary.

As for the user study, we do not believe it is necessary since the comments we use comments that developers use. For a user study, we would need to detect something that is not written by the developers so they can verify it (e.g., use metrics to detect TD), but that is not the case here. We should add a small part in the paper to address this.  
------------------------------------------------------------------------------------------------------------------

== Major comments to authors  
  
• (R2-4) It would be nice to have something more concrete on the relationship with [10]. For instance the process overview is nearly identical except for the label NLP classification. For historical record, it would be useful to have both studies amalgamated here (which I think is permissible given copyright). The most glaring omission in my view is the criteria by which some comment is classified as requirements vs design debt. I think readers would be curious to know how you distinguish between these two types, and from the other 3 types. (e.g. P25 of this submission). There is certainly some room for debate in how you are classifying them. Perhaps another option is to link to the training manual you provided coder 2 for his/her task.

Thank you for the comment. We agree with the reviewer that defining self-admitted technical debt and the criteria that we used when classifying them is a must, and that the journal paper should stand on its own instead of relying on references that could be ignored or even unpractical for the reader. Reviewer 1 had a similar comment (R1-10). Please refer to R1-10 or to section 2.4 paragraphs 3 – 6 in the manuscript.

• (R2-5) Finally, you repeatedly refer to study [10] as 'state-of-the-art' which is true, inasmuch as it represents the only other study to my knowledge approaching the issue of identifying TD through code comments. But the paper would be improved by merging the two papers, in my view, and I'm curious why you chose to divide them.

Thank you for the comment. The reference that the reviewer is mentioning is the previous work by Potdar and Shihab published on 2014. In their study the authors devised 62 comment patterns (i.e., words and short phases) from source code comments extracted from 5 open source projects. To date, these 62 comment patterns, are the “state-of-the-art” on detection of self-admitted technical debt. That said, we think that the reviewer is really talking about of our other work, by Maldonado and Shihab published on 2015 [11, in the original text], that has much more in common with the current work under revision than the first one.

That said, we initially decided to divide them since one was about the dataset and determining what types of self-admitted technical debt are most common, whereas this journal paper is more about using NLP to detect self-admitted technical debt.

Nevertheless, we considered what the reviewer is mentioning and merged some parts from the previous work to make the journal paper stand on its own. We consider that this was achieved by addressing comments R1-10 and consequently R2-4. We hope that the provided explanation addressed this issue, however, if it has not, we are happy to incorporate any other changes the reviewers suggest.

------------------------------------------------------------------------------------------------------------------

Sure, we can try to merge some parts of the two.

We initially decided to divide them since one was about the dataset and determining what types of TD are most common, whereas this paper is more about using NLP to detect SATD.

------------------------------------------------------------------------------------------------------------------

• (R2-6) F-measure is the harmonic mean, which implies both P and R are valued equally. However, there are good reasons for thinking this is not the right model for software problems (see e.g. Berry et al REFSQ 2012 "The case for dumb RE tools") and that instead recall should be the target. In this case, one use for the tool is to find code with technical debt. Would a dev rather see all the code with TD, at the expense of some more noise, or greatly reduce the noise and miss some actual TD? My instinct tells me the latter. I would like to see your view on the subject. In any case a naive 50/50 split like F1 seems incorrect. From Appendix table 7 it seems like your approach (in this paper and the previous one) favor high precision vs high recall. Can you explain why this is desirable? Perhaps the case made in Sadowski's ICSE2014 paper on industrial static analysis tools (now called Shipshape), namely, devs hate the noise.

Thank you for the comment. In Berry et al. “The Case for Dumb Requirement Engineering Tools” the authors explain how four different types of NLP tools are used to analyze requirement documents. Extracting requirements from documentation is considered as a tedious and error prone task for analysts, and the assistance of tools is highly appreciated. The authors argue that tools with high recall (few false negatives) is actually preferable than tools with high precision (few false positives). Their reasoning is that it is easier for an analyst to manually eliminate false positives identified by the tool than finding the possible missing false negatives by themselves. Therefore, a tool providing 100% recall would prevent a fully manual effort to extract requirement as the analyst should be concerned only on eliminating the false positives.

Berry et al. presents solid reasoning that makes perfect sense when extracting requirements from existing documentation. In this specific scenario it is necessary to identify all requirements that it is contained in the analyzed documents. However, when dealing with technical debt maters tend to be different, as the reviewer points out. In

A. Bessey et al. “A few billion lines of code later: Using static analysis to find bugs in the real world” the authors document important challenges that they faced while developing and commercializing bug finding tools. Among other things, they discuss the implications of having a high number of false positives, and how this is not desired. For example, they say that true bugs get lost in false bugs, people will ignore the tool and ultimately will lost trust on it. More recently, Ernest et al. “Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt” conducted a survey of 1,831 participants that included software engineers, architects and developers from three large organizations. They report that the adoption of tools and approaches to identify and manage technical debt is uncommon. One of the reasons is that false positives produces a lot of noise and it is cumbersome to deal with it. Sadowski et al. “Tricoder: Building a Program Analysis Ecosystem” offers many insights on why static analysis tools are often not used effectively in practice, and again, false positives appear as not desired result.

The way that we envision the approach being applied follows the line of thought of Bessey, Ernest and Sadowski which favors precision instead of recall. That said, we agree with the reviewer point that some readers cold benefit of an approach that favors recall. To address this comment we added the following text in the discussion section 4.3 paragraph 4.

“According to previous work, developers hate to deal with false positives results (i.e., low precision) [33],[34],[35]. Due to this fact, we choose to present our results in this study using the Logistical Regression algorithm, which has an average precision of 0.716 throughout all projects. However, results that favors recall over precision (e.g. Naives Bayes algorithm) might well be fine if a manual process to filter out false positives is in place, as reported by Berry et al. [36].”

The citation mentioned and that was used to add this text is the following:

[33]  A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, “A few billion lines of code later: Using static analysis to find bugs in the real world,” *Commun. ACM*, pp. 66–75, 2010.

[34]  N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, “Measure it? manage it? ignore it? software practitioners and tech- nical debt,” in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 50–60.

[35]  C. Sadowski, J. v. Gogh, C. Jaspan, E. Soderberg, and C. Win- ter, “Tricorder: Building a program analysis ecosystem,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineer- ing*, 2015, pp. 598–608.

[36]  D. Berry, R. Gacitua, P. Sawyer, and S. F. Tjong, *The Case for Dumb Requirements Engineering Tools*, 2012.

-------------------------------------------------------------------------------------------------------------------

Can we try to find some related work (and there is tons that interviews developers) to say what they prefer, precision or recall.

Yes, we can say this and cite the ICSE2014 work, or see if the Brazilian guys also have some motivation for higher precision.

------------------------------------------------------------------------------------------------------------------

• [EMAD](R2-7) The random classifier puzzles me. It sounds like you have it randomly bucketing something as TD based on the underlying model you derive from the manual labeling. E.g. if the source dataset had 6% TD, 6% of the time (randomly) this classifier assigns the TD label. But why should the random classifier have to know the underlying distribution? What would you get if you set it to 50%?

No, it would be 50% if the dataset was balanced – but in this case it clearly is not balanced. This is why we actually use the distribution in the data to know the accuracy of the random classifier. We should give an example here, e.g., red vs. blue balls in a bucket. We can also cite other work (preferably not ours) that did this.

• [EMAD] (R2-8) What I am saying is that the random classifier in your approach has this prior that in reality a naive classifier wouldn't get. And at 50% I suspect the recall would be much closer to the NLP approach. To be honest the discussion on page 6 was very unclear on how you ran this. I don't see why it needs some particular calculation for precision or recall, since you will simulate it just like a regular classifier, then measure the P/R based on the classification results.

No, it would be 50% if the dataset was balanced – but in this case it clearly is not balanced. This is why we actually use the distribution in the data to know the accuracy of the random classifier. We should give an example here, e.g., red vs. blue balls in a bucket. We can also cite other work (preferably not ours) that did this.

• (R2-9) Finally, I expected to see in the results something that looked at TD vs non-TD (that is, no categories). The reasoning for this is that assuming the categories are invalid, even knowing there is TD of some kind would be helpful; it would therefore be interesting to know if performance changes.

Thank you for the comment. We designed our experiment to distinguish between different types of self-admitted technical debt because it enables each type of debt to be handled in a specific way. However, we do agree with the reviewer point that it is worthwhile knowing the approach performance when identifying TD vs non-TD.

To address this comment we added subsection 4.2 under the Discussion section.

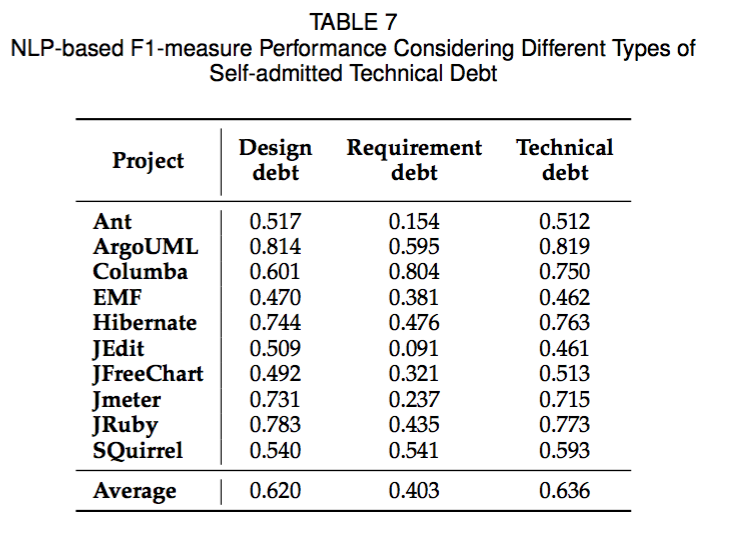
“**Distinguishing Self-Admitted Technical Debt from Non-Self-Admitted Technical Debt Comments**

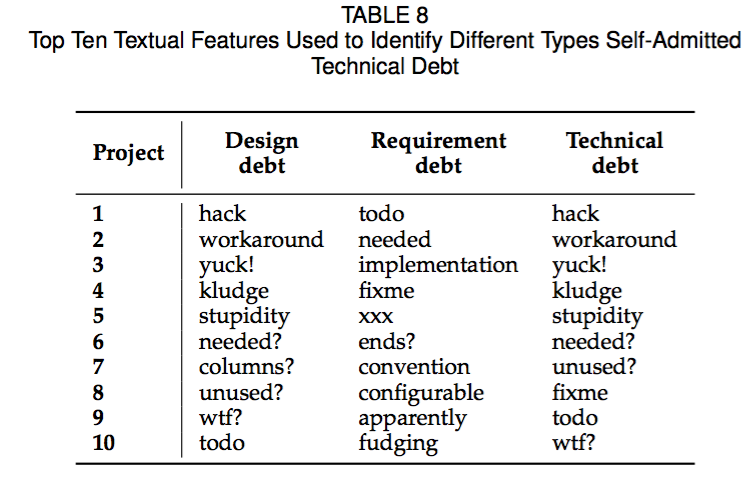
So far, we analyzed the performance of our NLP-based approach to identify distinct types of self-admitted technical debt (i.e., design and requirement debt). However, a simpler distinction such as self-admitted technical debt vs non-self-admitted technical debt can also be interesting in the case those specific categories are not needed. For example, if self-admitted technical debt is being used as a quality metric assessment, the user might not care about the particular type of debt that is present in the code, but for the total quantity of self-admitted technical debt that there is.

In order to compute the performance of our NLP-based approach in this new scenario we execute a similar experiment that in RQ1 and RQ2, but with modified training and test datasets. First, we took all design and requirement self-admitted technical debt comments and assigned them an unique type i.e., technical debt, and the remainder of comments we kept as without technical debt. Second, we run the maximum entropy classifier in a cross project basis, using the comments of 9 projects to train the NLP classifier and the remainder project comments to test the classification. We repeat this process for each of the ten projects, each time training on 9 projects and testing on the remaining 1 project. Lastly, we analyzed the textual features used to identify the self-admitted technical debt comments.

Table 7 compare the F1-measure between design debt, requirement debt and a combination of both. As we can see, the performance when identifying technical debt is very similar with the performance when identifying design debt. This is expected, as the majority of technical debt comments in the training dataset are indeed design debt. Nevertheless, the performance achieved when identifying design debt was surpassed in projects where our approach performed well identifying requirement debt, for example in Columba (0.601 to 0.750) and SQuirrel (0.540 to 0.593).

We find that the average performance using the combination of design and requirement self-admitted technical debt is higher (0.636) than the performance achieved when calculating them individually (0.620 and 0.403 for design and requirement debt respectively).





Also, when analyzing the top 10 textual features used to classify self-admitted technical debt, we find once more, a high similarity with the top 10 textual features used to identify design technical debt. The weigh of the features is attributed in accordance with the frequency that each word is found in the training dataset, and therefore, the top 10 features tend to be similar with the design debt ones as design debt comments represents the majority of self-admitted technical debt comments in the dataset. Table 8 shows a comparison of the top 10 textual features used to identify design, requirement and self-admitted technical debt comments.”

------------------------------------------------------------------------------------------------------------------

Sure, we can add an experiment that does TD vs. non-TD. The reason we initially did the categories since we thought it is the most informative option, but we do agree with the reviewers very good point that it is worthwhile knowing TD vs. non-TD.  
------------------------------------------------------------------------------------------------------------------

• (R2-10) I don't understand why in S 2.4, for the coding agreement portion, you selected a random dataset that nearly matched the breakdown of the real world datasets. This results in a very low number of actual true positives (e.g., only 1 comment of the 659 was doc debt). So when we calculate Kappa, my concern is that kappa here is really measuring a comment is "debt or non-debt", and not the specific categories. In other words it does not help us assess whether what coder 1 is calling "requirements debt" is also what coder 2 calls requirements debt. A mitigation here is to include category specific kappa scores, or to use a non-representative sample (I don't get why this is important to the task).

Thank you for the comment. Indeed, we kept the dataset as close as possible from a real world scenario. The intention here is to validate the dataset as a whole, decreasing the bias towards an specific type technical debt. For instance, before the classification, the student was not told the expected quantity of each debt in the dataset. This is a positive reinforcement that we were also able to identify what is not technical debt correctly. However, we agree with the reviewer that focusing only on different types of self-admitted technical debt is also a viable option. We also agree with the reviewers concern that in our case Kappa could be in reality measuring debt or non-debt comments. To address this comment we added/modified the text in section 2.4 and the threats to validity with category specific kappa scores for design and requirement self-admitted technical debt.

**Section 2.4, last paragraph:**

“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 +0.81 for requirement self-admitted technical debt. According to Fleiss [31] values larger than 0.75 are characterized as excellent.”

**Threats to validity, first paragraph:**

“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 [31] 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 much more without self-admitted technical debt comments than the other categories), 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.”

As part of this change, we also added the following citation:

[31] J. Fleiss, “The measurement of interrater agreement,” *Statistical methods for rates and proportions.*, pp. 212–236, 1981.

------------------------------------------------------------------------------------------------------------------

We kept the fraction of the selected data to be the same proportion as the real world dataset since we did not want to bias the outcome of the classification. The reviewer has a good point about the fact that we in essence are comparing whether there is TD or not, and not focusing on the type per se. What we can do is either give the category specific kappa value if we have it or re-do this with a non-representative sample.

------------------------------------------------------------------------------------------------------------------

• (R2-11) Your text frequently says "NLP classifier" (and you mean max entropy classifier). A decision tree would be interesting here, given your RQ2 and the problem of what words are important. I would think the decision tree could present this quite nicely. There are other classifiers to attempt too (and something like Weka would give you access to all of them and let you see which is most suitable). Other SE papers on NLP, e.g. Andi Marcus or Abram Hindle in topic modeling, discuss these extensively. The paper would benefit from presenting other ways of doing the classification. The state of the art in the SE/NLP world is moving to a deeper consideration of these questions - you would benefit from having discussions with the linguistics folks at your institution. (I see later you do discuss 2 others in Section 4.2. But now I see you discounting Naive Bayes although it does better on recall - see above for why this might well be fine). The related work covers some of the applicable work, but I'm not sure just because the subject matter is different - e.g. traceability vs self-admitted debt - that the underlying NLP approach is not still relevant for comparison. Some more discussion of your classifier and why it makes sense in the context of the domain is merited.

Thank you for the comment. Indeed, what we meant by “NLP classifier” was max entropy classifier. We modified all occurrences of “NLP classifier” in the manuscript to the more appropriate term as pointed out.

As the reviewer noticed, we discuss 2 other ways of doing the classification in the discussion subsection 4.3. Also, we agree with the reviewer that we were discounting Naïve Bayes because it performed poorly in precision. We believe that we addressed this comment by responding R2-6 where we present the case why developers prefers precision over recall, and by explaining the scenario where recall over precision could be interesting.

Also, for the remaining of the comment we agree that more discussion of our classifier is, indeed merited, and how to choose a specific classifier kind as well. To address this comment we added the following text in subsection 4.3 paragraph 5.

“One important question to ask when choosing what kind of classifier to use is how much training data is currently available. Must often, 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 learn theory recommends classifiers with high bias, such as Naive Bayes [37], [38]. If there is a reasonable amount of labeled data, then you are in good stand to use most kinds of classifiers [32]. For instance, you may wish to use an Support Vector Machine (SVM), a decision tree or, like in our study, a max entropy classifier. If a huge amount of data is available, then the choice of classifier probably has little effect on your results and the best choice may be unclear [39]. It may be best to choose a classifier based on the scalability of training or even runtime efficiency.”

As part of this change, we also added the following citations:

[32]  C. D. Manning, P. Raghavan, and H. Schutze, *Introduction to information retrieval*. Cambridge University Press, 2008.

[37] G. Forman and I. Cohen, “Learning from little: Comparison of classifiers given little training,” in Proc. PKDD, 2004, pp. 161–172

[38] A. Y. Ng and M. I. Jordan, “On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes.” in *Proc. NIPS*, 2001, pp. 841–848.

[39] M. Banko and E. Brill, *Scaling to Very Very Large Corpora for Natural LanguageDisambiguation*, 2001.

------------------------------------------------------------------------------------------------------------------

Let’s try to add a discussion here of the different underlying classifiers, and more importantly, why they work. I think that using decision trees would be crazy since we will have many branches, but I am not sure. Perhaps we can try that.

We should also re-word the naïve bayes so that it is not “discounted” and highlight its strengths.

------------------------------------------------------------------------------------------------------------------

• (R2-12) Re: the number of comments needed to classify, you say something like 1400 for design and 300 for requirements. Fine. But I notice that per-project the totals you have are much lower for overall TD comments. So are you implying that with only (say) Columba, I would not get a satisfactory F1 because the number of comments is too low? Perhaps you can suggest some mitigation to that, e.g. with transfer learning. Bigger question: what leads you to think a comment that is requirement debt in project A is also requirement debt in project B? My experience is that these projects are very context specific, particularly in requirements debt.

Thank you for the comment. We believe that some comments are specific and some are more general. Intuitively, requirement debt can be context specific as different projects have different needs. However, requirement debt often expresses the partial complement of requirements. For example, the following two requirement debt comments from different projects have many features (words) in common although they are context specific.

“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]

As the max entropy classifier utilizes a combination of many features to vote a comment into a class, these two comments have a greater probability to be assigned to the same class. Moreover, in the original manuscript we have a discussion section providing more details on the similarity between requirement debt comments.

------------------------------------------------------------------------------------------------------------------Yes, indeed we believe that some comments are specific and some are more general. The beauty of a NLP-based approach is that it can not only learn words, but also learn from the structure of the comments, hence it can (and as we show does) transfer. We should discuss this in the paper.

------------------------------------------------------------------------------------------------------------------  
  
• (R2-13) The RQ3 is really about active learning. There are in fact tools and approaches in this sub-field entirely concerned with reducing the amount of labeling needed. See <https://en.wikipedia.org/wiki/Active_learning_(machine_learning)>

Thank you for the comment. Indeed, it is possible that active learning could be used to reduce the amount of labeling needed. Tong and Koller in “Support Vector Machine Active Learning with Applications to Text Classification” states that in many supervised learning tasks, labeling instances to create a training dataset is time consuming and costly, and finding ways to minimize the number of labeled instances is often beneficial. Usually, the training dataset is chosen to be a random sampling of instances. However, in many cases active learning can be employed where the learner actively chooses the training data from an unlabeled pool that will be labeled on the fly. The expected result is that this extra flexibility will reduce the learner’s need for large quantities of labeled data.

However, in our case we are reporting on the performance of the maximum entropy classifier given a certain number of already labeled training data, which in the context of our study, makes sense. That said, we believe that techniques such as active learning can be analyzed and compared in future work.

To address this comment we added the following text in the Conclusions and Future Work section, paragraph 5.

“Another interesting topic that we plan to analyze 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 expand even further the horizon of projects that our approach can be applied.”

------------------------------------------------------------------------------------------------------------------

Should mention this in the related work section at the least.

------------------------------------------------------------------------------------------------------------------

• (R2-14) It seems like you cannot decide whether the dataset should be one giant bag of all 10 projects, or 1 project at a time (which suggests some sort of multilevel regression, but anyway). In evaluating amount of training data needed, I don't understand why you approach this on a per-project basis. Wouldn't it make more sense to either look at the incremental improvement overall, or only on one project? There seems to be a huge assumption that these projects have similar (identical) feature distributions. That seems dangerous; perhaps not in this case, where the projects are all open source, Java projects, but certainly when we broaden the developers involved, I would expect to see a lot of drift (for example, would a German project use words like "hack"?)

Thank you for the comment. Indeed we agree with the reviewer that we should complement the research question with a per-comment experiment. To address this comment we added/modified the text in RQ3 starting on paragraph 2 as follow:

“**Approach**: To answer our research question, we conducted two different experiments. On both of them we followed a systematic process where we incrementally add training data and evaluate the performance of the classification. However, for the first experiment we added training data in a per-project basis (i.e., a variable number of comments all belonging to the same project at the same time), whereas for the second experiment, we added data 100 comments at a time regardless of the project that it belongs.

To execute the first experiment, we use one project as testing data, and the remaining nine projects to train, since we have 10 projects in our dataset. However, we do not train the maximum entropy classifier with all nine projects, instead, we add each project incrementally. We repeated this process for each project and report on our findings.

It is important to notice that the order we added each project was not aleatory. After some experimentation we decided that the best way to train the max entropy classifier is to add first to the training dataset projects with more of the self-admitted technical debt comments being identified. This way we could achieve better classification performance quicker. Furthermore, adding projects in a randomly fashion would impact the classification performance and prevent us to properly analyze the results of the first experiment.

To determine how much data is required to effectively identify self-admitted technical debt comments, we compute the F1-measure after each iteration (an iteration is simply a run with a different size of training data). We record the iteration that achieves the highest F1-measure and the number of projects used in the training dataset to achieve this F1-measure. Then, we record the number of projects needed to achieve at least 90% and 80% of the maximum F1-measure.

For example, if the maximum F1-measure is 0.85 and it is achieved in the eighth iteration (i.e., using 8 projects in the training dataset), and during the fourth iteration we achieve a F1-measure of 0.80, then we say that we can achieve at least 90% (94% to be exact) of the maximum F1-measure with a training dataset constructed from just 4 projects. Since the results will differ for the different projects, we repeat this analysis for all projects and present the average F1-measures across all projects.

To execute the second experiment, we shuffle the comments from all projects into a big dataset. Then, we split this dataset into 10 identical parts making sure that each part has an identical ratio of self-admitted technical debt and without technical debt comments. Next, we use one of the ten parts to test the max entropy classifier and the other 9 to train it. The training data is incrementally feed into the classifier by batches of 100 comments at time, and we also make sure that each batch of 100 comments contains the same ratio of self-admitted technical debt and without technical debt comments. We repeated this process for each one of the 10 dataset parts and report on our findings.

For this experiment, to determine how much data is required to effectively identify self-admitted technical debt comments we also compute the F1-measure after each iteration and record the iteration that achieves the highest F1-measure. However, instead of recording the number of projects that were used to train the dataset we report the number of comments that was used to achieve this F1-measure. The number of comments is computed in multiples of 100, due to the size of each batch added to the training dataset.

**Per-project based experiment results - design debt:** Figure 3(a) shows the F1-measure using different sizes of training data for the Ant project. Due to space, we discuss the results for a representative project (Ant) in this section, however, figures for all projects are provided in the Appendix (Figures 7 – 24). From Figure 3(a), we find that the maximum F1-measure improves as we increase the number of projects (i.e., iterations), achieving the highest F1-measure in the seventh iteration and slightly decreasing afterwards. The horizontal lines in the figure show the 80% and 90% of the highest F1-measure. We can see from Figure 3(a) that with 1,499 comments (i.e., from 3 projects) and 1,815 comments (i.e., from 4 projects), we can achieve 80% and 90% of the highest F1-measure, respectively. This amounts to a reduction of 37.6% and 24.5% in training data to achieve 80% and 90% of the maximum F1-measure, respectively. Considering the tradeoff in accuracy versus the amount of training data, for Ant, using only 3 or 4 projects provides the best tradeoff.

We analyzed all iterations from all projects to determine the iterations that achieve the best F1-measure performance. To measure that, we calculate the average percentage of the maximum F1-measure for each iteration. For example, we take the average percentage of the maximum F1-measure achieved during the first iteration for all projects, then we calculate the same value for all second iterations and so on. We find that, the best performance is achieved during the eighth iteration, with an average maximum F1-measure of 96.57% using (on average) 2,353 comments to create the training dataset. In comparison, the ninth iteration has an average maximum F1-measure of 95.99%, which is slightly lower than the average obtained in the eighth iteration, and uses more comments in the training dataset (i.e., 2,432).

Table 5 shows the average percentage of the maximum F1-measure for each iteration. The first column shows the iteration number. The second column shows the average percentage of the maximum F1-measure achieved for each iteration. The third column presents the delta of the average percentage of the maximum F1-measure between one iteration and the previous one. The fourth column shows the average number of comments used in the training dataset of that specific iteration. From Table 5 we observe that, on average, we can achieve more than 80% and 90% of the maximum F1-measure in the second and third iterations, respectively. To achieve more than 80% and 90% of the maximum F1-measure, we require 1,106 (49.13% of total comments required in seventh iteration) and 1,444 (64.14% of total comments required in seventh iteration) comments, respectively. Moreover, we see that the second and third iterations provide the highest delta between iterations.

**Per-comment based experiment results - design debt:** Figure 4(a) shows the ten fold average F1-measure obtained while identifying design self-admitted technical debt. We find that the maximum F1-measure improves as we increase the number of comments in the training dataset, achieving its highest value (i.e., 0.824) with 42,700 comments. However, the most accentuated improvement in the F1-measure performance happens within the first 10k comments, more precisely 80% and 90% of the maximum F1-measure is achieved with 3,900 and 9,700 comments in the training dataset. In this experiment we added 100 comments per iteration, and for each batch of 100 comments we keep the same ratio between design self-admitted technical debt and without technical debt comments of the training dataset which is approximately 5% (i.e., 2,703/58,122. Therefore, we used 195 and 485 design self-admitted technical debt comments to achieve 80% and 90% of the maximum F1-measure respectively. This amounts to a reduction of 90.86% and 77.28% in training data to achieve 80% and 90% of the maximum F1-measure.

The number of design self-admitted technical debt comments needed to achieve 80% and 90% of the maximum F1-measure in our second experiment is considerable lower than the number of self-admitted technical debt comments necessary to achieve similar marks on the first experiment. One possible reason is that in the second experiment (i.e., per-comment based) comments that belongs to same project are used as training data for the maximum entropy classifier.

**Per-project based experiment results - requirement debt:**  We find that, although there is a variation in the F1-measure value during the first 3 iterations, they are not as preeminent as the variation found in design self-admitted technical debt analysis. The F1-measure in requirement self-admitted technical debt tends to be more constant through the iterations, and the first iteration has a high percentage of the maximum F1-measure achieved for each project. This shows that the way developers indicate requirement debt does not vary between different application domains as much as in design debt. This uniformity in requirement self-admitted technical debt comments allows a good classification even with a small number of comments in the training dataset. We elaborate more on this point later in Section 4.

Figure 3(b) shows the F1-measure for different iterations in ArgoUML. The highest F1-measure of 0.65 is achieved in the third iteration. From Figure 3(b), we observe that we achieve more than 80% of the highest F1-measure in the first iteration and more than 90% in the second and subsequents iterations. The reduction in comments is 28.64% and 62.31% for the 90% and 80% of the maximum F1-measure, respectively.

Table 6 shows the average percentage of the maximum F1-measure for each iteration. Unlike the case of design debt, for requirement debt, the best F1-measure is achieved in the first iteration. This shows that using as few as 380 comments, we can effectively detect requirement self-admitted technical debt.

**Per-project based experiment results - requirement debt:** Figure 4(b) shows the ten fold average F1-measure obtained while identifying requirement self-admitted technical debt. As intuitively expected, the F1-measure increases as we add more data into the training dataset, and again the biggest improvement happens around the first 10k of comments. For requirement self-admitted technical debt we achieved 80% of the maximum F1-measure using 2,600 comments and 90% of the maximum F1-measure with 11,800. During this experiment we increase the size of the training dataset 100 comments per time, and within each batch of 100 comments we added 2 requirement self-admitted technical debt comments to keep the same approximate ratio of the total requirement self-admitted technical debt and without technical debt comments (i.e., 757/58,122) until we have added all requirement self-admitted technical debt comments available to train. While training the dataset with 2,600 comments (80% of the maximum F1-measure) we are in fact adding 52 requirement self-admitted technical debt comments, whereas to achieve 90% of the maximum F1-measure we used 236 requirement self-admitted technical debt comments. The maximum F1-measure achieved was 0.753 using 51,300 comments of which 675 were requirement self-admitted technical debt, and therefore, this amounts to a reduction of 92.29% and 65.03% in training data to achieve 80% and 90% of the maximum F1-measure.”

Also, for the remainder of the comment we agree with the reviewer that there are limitations to the generalization of our approach. We believe that we had already addressed the issue of generalization on the Threats to Validity while responding R1-4.

“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 that are written on a different idiom than English.”

------------------------------------------------------------------------------------------------------------------

Yes, that is a good point and we should mention this in the threats (if we do not do that already). As for the point about per-project, we try to make the scenario as realisitic as possible, therefore, we envision that people will not give one comment but rather classify an entire project and train on it since we need positive and negative instances.

------------------------------------------------------------------------------------------------------------------

• (R2-15) Furthermore, could you be explicit (or rephrase, if I missed it) whether the analysis changes the ordering of the projects? For example, since Ant has a lot of examples, using that first I would expect would be more useful than Columba. This is another argument for merging all the labeled features into one giant bag, and pulling them out comment by comment (or 10/100 whatever).

Thank you for the comment. The reviewer is correct the information is not clear in the original manuscript. In addition, the analysis changes accordingly with the selected ordering that the max entropy classifier is trained, but not the final result (i.e., when all projects are already added to the dataset)

To address this comment we added the following text in RQ3, paragraph 2.

“It is important to notice that the order we added each project was not aleatory. After some experimentation we decided that the best way to train the max entropy classifier is to add first to the training dataset projects with more of the self-admitted technical debt comments being identified. This way we could achieve better classification performance quicker. Furthermore, adding projects in a randomly fashion would impact the classification performance and prevent us to properly analyze the results of the first experiment.”

-------------------------------------------------------------------------------------------------------------------

We do not do that, we use the project with the most comments first and go down the line. We should do an experiment where we put all comments in one line and add 10 or 50 comments per time and see what is the optimal number of comments to train on.  
------------------------------------------------------------------------------------------------------------------

• (R2-16) You don't really discuss contruct validity, which I think is key to your work. Namely, does the construct of 'self-admitted TD' match with what is actually TD? In other words, does labeling a comment TD imply that the code following is actually TD (since ultimately managers etc care about the code). You could determine this by either inspecting the code history (to show the comments are still relevant), or by running some of the code smell work against the sections of code you identified to see how much overlap there is (ideally, 100% of the fragments you identify are found by a TD code smell detector, but that won't be the case).

Thank you for the comment. Indeed, we agree with the reviewer that the original manuscript was lacking discussion on construct validity. We also agree that it is important to understand how self-admitted technical debt relates to “actual” technical debt in the source code. That said, we believe that the experiment that we conducted to address comment R2-3 also address what the reviewer is pointing out here, as we could shed some light on how self-admitted technical debt and code smells overlaps in the analyzed projects.

Also, concerning the relevancy of code comments, Fluri et al. [40] analyzed the co-evolution of source code and code comments, and found that 97% of the comment changes are consistent. In addition to that, Potdar and Shihab [14] analyzed if the developers update the source code containing self-admitted technical debt. They inspected the source code files to determine the frequency of four possible cases: Case 1 the self-admitted technical debt was removed along with change in enclosing code; Case 2 the self-admitted technical debt was removed but enclosing code was unchanged; Case 3 the self-admitted technical debt persisted despite enclosing code changing; Case 4 the self-admitted technical debt persisted with no change in enclosing code. They found that inconsistent changes (Case 2 and 3) happened only in 8.8% of the time in one of the analyzed project and that across all analyzed projects inconsistent changes are minority.

Other aspects of the construct validity we believe that was addressed by responding the following review comments: R1-5, R1-13, R1-14 and R2-10.

-------------------------------------------------------------------------------------------------------------------

We should mention this in the threats to construct validity, however, we do not expect an overlap since this is self admitted TD. We can do a small experiment to validate this and/or cite the ICSME paper which says that code and comments co-change most of the time.  
------------------------------------------------------------------------------------------------------------------

• (R2-17) I think the internal validity is otherwise ok, I appreciated you used another rater. I think my comments on other NLP approaches above also touch on improving internal validity (e.g., recall over precision).

Thank you for the comment. We believe that we addressed this comment by the changes made on comment R2-10.

== Minor  
- could you report totals/avg overall in Table 1? Also percentage of TD comments might be helpful in addition to absolute #  
- in S2.4, does the previous study use the same projects as this study?  
- Section 3 is called "Case Study Results". This isn't a case study but rather an experiment.  
- You cite the Stanford NLP tools [15], but you aren't using them (AFAIK). You are using instead the max entropy classifier at <http://nlp.stanford.edu/software/classifier.html>, right? That should be cited instead.

Sure, we should address all these.

**Reviewer: 3**  
Public Comments (these will be made available to the author)  
In this paper, the authors propose an approach to determine self-admitted technical debt (i.e., code improvements) based on comments in the source code using natural language processing techniques and machine learning. They apply their approach to 10 open source projects.  
  
The work is very interesting, and I enjoyed reading about the methodology for identifying technical debt comments. However, I have concerns about how this work will be used and some possible threats to validity:  
  
• (R3-1) I readily agree that technical debt is an important topic, but it’s not clear how important detecting it is — what actions can be taken that could improve software quality or a software’s lifecycle in some way? I would like to see the authors provide further motivation on how their approach could be used.

Thank you for the comment. During the past couple of years technical debt has been gaining traction from software engineers practitioners and from the research community as well. The metaphor ability to communicate intrinsic technical problems in software projects in a way that is clear and understandable to non-technical people has proved to be one of the metaphors biggest strength. Moreover, the software community is taking advantage from this financial metaphor and developers and management alike take on “debt” willingly by choosing to implement non-optimal solutions to achieve short terms goals. However, to enable this risk mechanism to be effectively applied some mitigations has to be in place first. The most important step to manage technical debt is to be able to properly find it. Once technical debt is identified it can be monitored and removed as necessary. More recently, a large scale survey stated how the current tools are, in their opinion, not satisfactory to properly manage technical debt [34]. Our approach can help fill the gap of ineffective tools helping practitioners to maintain and improve software projects.

That said, to address this comment we added the following text that was also pointed out in R1-3 in the Introduction paragraph 7.

“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 well familiar with good design and coding practices. 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 is definitely more relevant to them.”

-------------------------------------------------------------------------------------------------------------------

We should mention that once the TD is identified, then it can be managed or removed. We should cite some of the current work that says TD is unavoidable, and needs to be managed.

-------------------------------------------------------------------------------------------------------------------

• (R3-2) Did the authors do any vetting on their development set of whether or not the debt is actually present in the code when it is in the comment? In essence, are they just finding comments that \*admit\* technical debt, or are they actually finding locations in the code that actually \*have\* technical debt? For instance, did the authors consider the impact of obsolete comments? If a developer eliminated the technical debt, would the comment indicating the debt remain?

Thank you for the comment. Indeed, we agree with the reviewer that verifying the relation between the source code comment and the code it self it is a must. Reviewer 2 had made a similar comment in R2-3. To address this first part of the comment we tackled the problem by conducting an experiment that we compare the overlap between our source comment based approach with a static analysis tool to detect bad smells at the file level. We added the text that report on our findings in section 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, for example architectural reviews, documentation analysis and static analysis tools. To date, using static analysis tools is the most traditional approach to identify technical debt in the source code. In general, static analysis tools parse the source code of a project and calculate metrics and thresholds to identify possible object oriented design violations, also know as bad smells or technical debt.

In this subsection we analyze the overlap between what our NLP-based approach identifies as technical debt and what a static analysis tool identifies as technical debt. JDeodorant is our static analysis tool of choice as it is capable of identify design debts (i.e., bad smells) in Java projects, and suggest refactoring opportunities to solve them.

First, we analyzed our 10 open source projects using the static analysis tool. The result of this analysis is a list of Java files that were identified with at least one bad smell. In this process we aimed to identify three common bad smells namely, long method, feature envy and god class. Second, we created a similar list containing the files that were identified with self-admitted technical debt comments. Lastly, we analyze the overlapping files and present results.

Table 9 provides details about each of the projects used in our study. The columns of Table 9 present the total number of files with self-admitted technical debt, followed by the number of files containing self-admitted technical debt comments and long method at the same time, the percentage of files that it represents from the total files with self-admitted technical debt, the number of files containing self-admitted technical debt and the bad smell feature envy, the percentage of files with self-admitted technical debt that this number of files represents, the overlapping number of files between self-admitted technical debt and god class and its percentage from the total of files with self-admitted technical debt, and finally, the total number of files containing both self-admitted technical debt and any type of bad smells as well its percentage from the total files with self-admitted technical debt.

Jmeter, for example, has 200 files that contains self-admitted technical debt comments, and 143 of these files also contains the long method bad smell (i.e., 71.5%). In addition, we can see that 20.5% of the files that has self-admitted technical debt contains feature envy and 48.5% of them possesses the god class bad smell. In summary we see that 80.5% of all files that contains a bad smell also contains self-admitted technical debt comments.

We find that the bad smell that overlaps the most with self-admitted technical debt is long method. Intuitively, this is expected as long methods are a very common bad smell and it tends to happen frequently. The overlap between files with self-admitted technical debt and long method ranged from 43.6% to 82% of all files containing self-admitted technical debt comments, and considering all projects, the average overlap is of 65%. In addition, 44.2% of god class files also contains self-admitted technical debt comments, and for feature envy, the average percent of files with self-admitted technical debt is of 20.7%. Taking all bad smells analyzed in consideration we find that, on average, 69.7% of files containing self-admitted technical debt also contain a bad smell.”



Also for the remainder of the comment we based ourselves on previous work that analyzes the change correlation between comments and source code. Fluri et al. [40] analyzed the co-evolution of source code and code comments, and found that 97% of the comment changes are consistent, as the reviewer also points out. In addition to that, Potdar and Shihab [14] analyzed if the developers update the source code containing self-admitted technical debt. They inspected the source code files to determine the frequency of four possible cases: Case 1 the self-admitted technical debt was removed along with change in enclosing code; Case 2 the self-admitted technical debt was removed but enclosing code was unchanged; Case 3 the self-admitted technical debt persisted despite enclosing code changing; Case 4 the self-admitted technical debt persisted with no change in enclosing code. They found that inconsistent changes (Case 2 and 3) happened only in 8.8% of the time in one of the analyzed project and that across all analyzed projects inconsistent changes are minority. That said, although remote there is still a possibility that obsolete comments could impact our results. To address this comment we added the following text into the Threats to Validity section, paragraph 4:

“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 to changes on comments [14], [40]. In addition, it is possible that a variety of technical debt that is not self-admitted is present in the analyzed projects. However, considering all technical debt is out of the scope of this work.”

These are the citations related to this change:

[14]  A. Potdar and E. Shihab, “An exploratory study on self-admitted technical debt,” in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 91–100.

[40] B. Fluri, M. Wursch, and H. Gall, “Do code and comments co-evolve? on the relation between source code and comment changes,” in *Proceedings of the 14th Working Conference on Reverse Engineering*, 2007, pp. 70–79.

-------------------------------------------------------------------------------------------------------------------

We should cite the ICSME paper or do our own analysis here of comment and code co-change.

-------------------------------------------------------------------------------------------------------------------

• (R3-3) This idea is related to my first point; I can envision some scenarios where detecting admitted technical debt might be useful (such as quick statistics for software quality), and where detecting actual technical debt might be valuable (e.g., to target maintenance efforts).  
  
There is related work to support that 97% of comment changes are consistent, which would go a long way to sidestep the obsolete comment issue. However, it is possible that the remaining 3% could in theory represent a significant sample of TD comments, given how few comments include technical debt. I would recommend the authors manually evaluate a small, representative random sample to see if this is an issue.  
  
I believe the work in its current form identifies technical debt comments. I think the paper would benefit from a deeper discussion of the potential benefits of this information independent of knowing whether there actually \*is\* technical debt present.

Thank for your comment. Indeed, we agree with the reviewer suggestions. We believe that this comment has been address during the changes made while answering comments R3-1 and R3-2.

We hope that the provided explanation and the modified text address this issue, however, if it has not, we are happy to incorporate any other changes the reviewer suggest.

-------------------------------------------------------------------------------------------------------------------

Sure, we can do this and also point to the ICSME paper.

-------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------------

Is this the same as the first point? I don’t understand what he/she is asking here.

-------------------------------------------------------------------------------------------------------------------

• (R3-4) My final concern is with RQ 3 in determining the necessary amount of training data. Ideally, this RQ should be answered with random samples of different sizes from all projects. Did the authors at least try every possible ordering of the projects to identify how many projects were needed to reach within 10% of the highest F1 measure? It’s unclear in the text exactly what methodology was used, but the graphs in the appendix on p. 16 seem to indicate that they tried many different combinations (I just can’t determine if the order of projects was random).

Thank for your comment. We agree with the reviewer that it was not clear how we ordered the projects when adding them to the training dataset. Indeed we tried out many different methodologies, but we noticed that providing more training data to the maximum entropy classifier makes it reach the performance summit sooner. Therefore, we decided to add first projects that have more technical debt comments.

Also, we agree that to properly answer RQ3 we should consider comments from all projects and analyze the classifier performance incrementally. To address this comment (and also comment R2-14) we changed RQ3 by adding an experiment that shuffle all comments into one big dataset and them we incrementally (100 comments per time) increase the training dataset. Then, we report our findings in the following text added in RQ3.

“**Approach**: To answer our research question, we conducted two different experiments. On both of them we followed a systematic process where we incrementally add training data and evaluate the performance of the classification. However, for the first experiment we added training data in a per-project basis (i.e., a variable number of comments all belonging to the same project at the same time), whereas for the second experiment, we added data 100 comments at a time regardless of the project that it belongs.

To execute the first experiment, we use one project as testing data, and the remaining nine projects to train, since we have 10 projects in our dataset. However, we do not train the maximum entropy classifier with all nine projects, instead, we add each project incrementally. We repeated this process for each project and report on our findings.

It is important to notice that the order we added each project was not aleatory. After some experimentation we decided that the best way to train the max entropy classifier is to add first to the training dataset projects with more of the self-admitted technical debt comments being identified. This way we could achieve better classification performance quicker. Furthermore, adding projects in a randomly fashion would impact the classification performance and prevent us to properly analyze the results of the first experiment.

To determine how much data is required to effectively identify self-admitted technical debt comments, we compute the F1-measure after each iteration (an iteration is simply a run with a different size of training data). We record the iteration that achieves the highest F1-measure and the number of projects used in the training dataset to achieve this F1-measure. Then, we record the number of projects needed to achieve at least 90% and 80% of the maximum F1-measure.

For example, if the maximum F1-measure is 0.85 and it is achieved in the eighth iteration (i.e., using 8 projects in the training dataset), and during the fourth iteration we achieve a F1-measure of 0.80, then we say that we can achieve at least 90% (94% to be exact) of the maximum F1-measure with a training dataset constructed from just 4 projects. Since the results will differ for the different projects, we repeat this analysis for all projects and present the average F1-measures across all projects.

To execute the second experiment, we shuffle the comments from all projects into a big dataset. Then, we split this dataset into 10 identical parts making sure that each part has an identical ratio of self-admitted technical debt and without technical debt comments. Next, we use one of the ten parts to test the max entropy classifier and the other 9 to train it. The training data is incrementally feed into the classifier by batches of 100 comments at time, and we also make sure that each batch of 100 comments contains the same ratio of self-admitted technical debt and without technical debt comments. We repeated this process for each one of the 10 dataset parts and report on our findings.

For this experiment, to determine how much data is required to effectively identify self-admitted technical debt comments we also compute the F1-measure after each iteration and record the iteration that achieves the highest F1-measure. However, instead of recording the number of projects that were used to train the dataset we report the number of comments that was used to achieve this F1-measure. The number of comments is computed in multiples of 100, due to the size of each batch added to the training dataset.

**Per-project based experiment results - design debt:** Figure 3(a) shows the F1-measure using different sizes of training data for the Ant project. Due to space, we discuss the results for a representative project (Ant) in this section, however, figures for all projects are provided in the Appendix (Figures 7 – 24). From Figure 3(a), we find that the maximum F1-measure improves as we increase the number of projects (i.e., iterations), achieving the highest F1-measure in the seventh iteration and slightly decreasing afterwards. The horizontal lines in the figure show the 80% and 90% of the highest F1-measure. We can see from Figure 3(a) that with 1,499 comments (i.e., from 3 projects) and 1,815 comments (i.e., from 4 projects), we can achieve 80% and 90% of the highest F1-measure, respectively. This amounts to a reduction of 37.6% and 24.5% in training data to achieve 80% and 90% of the maximum F1-measure, respectively. Considering the tradeoff in accuracy versus the amount of training data, for Ant, using only 3 or 4 projects provides the best tradeoff.

We analyzed all iterations from all projects to determine the iterations that achieve the best F1-measure performance. To measure that, we calculate the average percentage of the maximum F1-measure for each iteration. For example, we take the average percentage of the maximum F1-measure achieved during the first iteration for all projects, then we calculate the same value for all second iterations and so on. We find that, the best performance is achieved during the eighth iteration, with an average maximum F1-measure of 96.57% using (on average) 2,353 comments to create the training dataset. In comparison, the ninth iteration has an average maximum F1-measure of 95.99%, which is slightly lower than the average obtained in the eighth iteration, and uses more comments in the training dataset (i.e., 2,432).

Table 5 shows the average percentage of the maximum F1-measure for each iteration. The first column shows the iteration number. The second column shows the average percentage of the maximum F1-measure achieved for each iteration. The third column presents the delta of the average percentage of the maximum F1-measure between one iteration and the previous one. The fourth column shows the average number of comments used in the training dataset of that specific iteration. From Table 5 we observe that, on average, we can achieve more than 80% and 90% of the maximum F1-measure in the second and third iterations, respectively. To achieve more than 80% and 90% of the maximum F1-measure, we require 1,106 (49.13% of total comments required in seventh iteration) and 1,444 (64.14% of total comments required in seventh iteration) comments, respectively. Moreover, we see that the second and third iterations provide the highest delta between iterations.

**Per-comment based experiment results - design debt:** Figure 4(a) shows the ten fold average F1-measure obtained while identifying design self-admitted technical debt. We find that the maximum F1-measure improves as we increase the number of comments in the training dataset, achieving its highest value (i.e., 0.824) with 42,700 comments. However, the most accentuated improvement in the F1-measure performance happens within the first 10k comments, more precisely 80% and 90% of the maximum F1-measure is achieved with 3,900 and 9,700 comments in the training dataset. In this experiment we added 100 comments per iteration, and for each batch of 100 comments we keep the same ratio between design self-admitted technical debt and without technical debt comments of the training dataset which is approximately 5% (i.e., 2,703/58,122. Therefore, we used 195 and 485 design self-admitted technical debt comments to achieve 80% and 90% of the maximum F1-measure respectively. This amounts to a reduction of 90.86% and 77.28% in training data to achieve 80% and 90% of the maximum F1-measure.

The number of design self-admitted technical debt comments needed to achieve 80% and 90% of the maximum F1-measure in our second experiment is considerable lower than the number of self-admitted technical debt comments necessary to achieve similar marks on the first experiment. One possible reason is that in the second experiment (i.e., per-comment based) comments that belongs to same project are used as training data for the maximum entropy classifier.

**Per-project based experiment results - requirement debt:**  We find that, although there is a variation in the F1-measure value during the first 3 iterations, they are not as preeminent as the variation found in design self-admitted technical debt analysis. The F1-measure in requirement self-admitted technical debt tends to be more constant through the iterations, and the first iteration has a high percentage of the maximum F1-measure achieved for each project. This shows that the way developers indicate requirement debt does not vary between different application domains as much as in design debt. This uniformity in requirement self-admitted technical debt comments allows a good classification even with a small number of comments in the training dataset. We elaborate more on this point later in Section 4.

Figure 3(b) shows the F1-measure for different iterations in ArgoUML. The highest F1-measure of 0.65 is achieved in the third iteration. From Figure 3(b), we observe that we achieve more than 80% of the highest F1-measure in the first iteration and more than 90% in the second and subsequents iterations. The reduction in comments is 28.64% and 62.31% for the 90% and 80% of the maximum F1-measure, respectively.

Table 6 shows the average percentage of the maximum F1-measure for each iteration. Unlike the case of design debt, for requirement debt, the best F1-measure is achieved in the first iteration. This shows that using as few as 380 comments, we can effectively detect requirement self-admitted technical debt.

**Per-project based experiment results - requirement debt:** Figure 4(b) shows the ten fold average F1-measure obtained while identifying requirement self-admitted technical debt. As intuitively expected, the F1-measure increases as we add more data into the training dataset, and again the biggest improvement happens around the first 10k of comments. For requirement self-admitted technical debt we achieved 80% of the maximum F1-measure using 2,600 comments and 90% of the maximum F1-measure with 11,800. During this experiment we increase the size of the training dataset 100 comments per time, and within each batch of 100 comments we added 2 requirement self-admitted technical debt comments to keep the same approximate ratio of the total requirement self-admitted technical debt and without technical debt comments (i.e., 757/58,122) until we have added all requirement self-admitted technical debt comments available to train. While training the dataset with 2,600 comments (80% of the maximum F1-measure) we are in fact adding 52 requirement self-admitted technical debt comments, whereas to achieve 90% of the maximum F1-measure we used 236 requirement self-admitted technical debt comments. The maximum F1-measure achieved was 0.753 using 51,300 comments of which 675 were requirement self-admitted technical debt, and therefore, this amounts to a reduction of 92.29% and 65.03% in training data to achieve 80% and 90% of the maximum F1-measure.”

-------------------------------------------------------------------------------------------------------------------

The order of the projects was not random, we put in the largest project first. Our analysis for reviewer 2 should address this comment as well.  
-------------------------------------------------------------------------------------------------------------------

• (R3-5) My issue with the conclusions the authors are making is all about the number of \*comments\* are needed, where in fact the only variable they are changing is the number of \*projects\*. Thus, I would tone down the claims in RQ3 to discuss number of projects, rather than number of comments, because it’s possible that a different ordering of projects would lead to a different conclusion in terms of comments. For example, maybe training can take place on 1 large project or 2 different projects with fewer comments and reach the same results with different numbers of comments. Ideally, I would have preferred to see more data points in this section before drawing conclusions about the number of comments needed. In fact, I would suggest the authors remove the data for Comment patterns & the random classifier and instead report line graphs for the NLP-based approach only, condensed into fewer graphs (one for each kind of debt).  
  
-------------------------------------------------------------------------------------------------------------------

Sure, we can tone down and discuss per project. I don’t think it is a good idea to remove the random and comment comparison.

-------------------------------------------------------------------------------------------------------------------

Specific comments:  
  
• (R3-6) - Table 2: I recognize that the data set is small, but I think it would be worth reporting some statistical analysis. The relationships seem quite pronounced and might still be statistically significant.

Thank you for the comment, Indeed we agree with the reviewer that reporting statical analysis would be interesting. To address this comment we added the following text into the original manuscript in Section 3, RQ1, paragraph 10.

“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<0.5) for both baselines and both design and requirement self-admitted technical debt.”

-------------------------------------------------------------------------------------------------------------------

Sure  
-------------------------------------------------------------------------------------------------------------------

• (R3-7) - RQ 2: I notice that some textual features include punctuation, while others don’t. Should punctuation be separated out? Is it the presence of the question mark alone in “needed?” or the word \*with\* the question mark that indicates technical debt? A brief justification of the author’s handling of punctuation would help. Did the authors try just “?” alone in predicting technical debt? I notice a number of features include a

question mark. Did the authors try technical features with & without the punctuation, or with the punctuation separated out to see the impact it would have?

Thank you for the comment. We did a lot of experimentation with the training dataset to understand which methodology would be the best fit for our approach. First, we tried to use comments as they were extracted. However, these comments were generating a lot of noise to the classifier that ended up hindering the performance of the classification. We removed then the unnecessary spacing (like tabs or new lines) and also Java comment syntax characters (//, /\* and \*/), we also had to remove punctuation such as ‘,’ or ‘.’ and finally we made everything lowercase as we explain in Section 2.5 on the original manuscript. That said, we choose to keep interrogation and exclamation points as they have a lot of meaning, and as we could observe during the manual classification, they can change the understanding of the comment. Also, these punctuations often helped to determine a self-admitted technical debt. To address this comment we modified the following text in Section 2.5 paragraph 5.

“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, 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 points. These specific punctuation was very useful during the identification of self-admitted technical debt comments, and provides insightful information about the meaning of the features.”

-------------------------------------------------------------------------------------------------------------------

Is this a phenomena of the NLP tool?  
-------------------------------------------------------------------------------------------------------------------

• (R3-8) - section 7: I think there needs to be a new paragraph introduced on line 44 at “Then,”

Thank you for your comment. Indeed, a new paragraph on line 44 increases the readability of the manuscript. As suggested by the reviewer we added the new paragraph in Section 7, paragraph 4.

“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 even using a low number of self-admitted technical debt comments in the training dataset can achieve high classification performance. 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 1,444 design self-admitted technical debt comments 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 380 self-admitted technical debt comments it is possible to classify with success requirement self-admitted technical debt automatically.”

-------------------------------------------------------------------------------------------------------------------

Sure

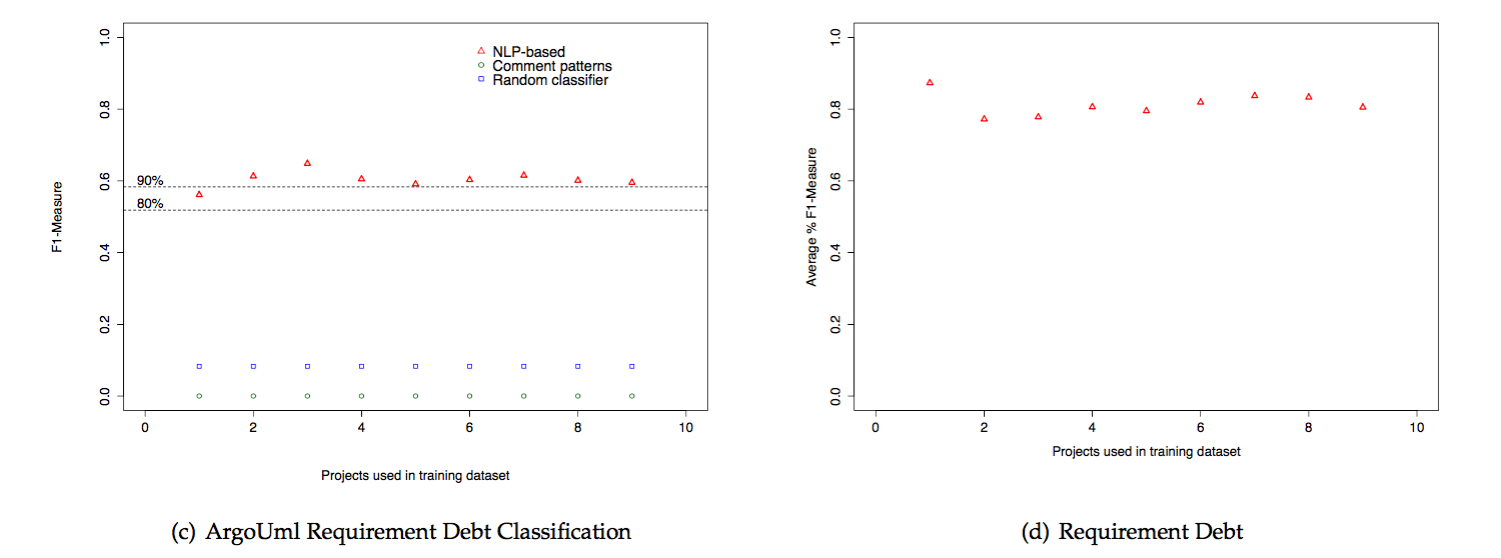
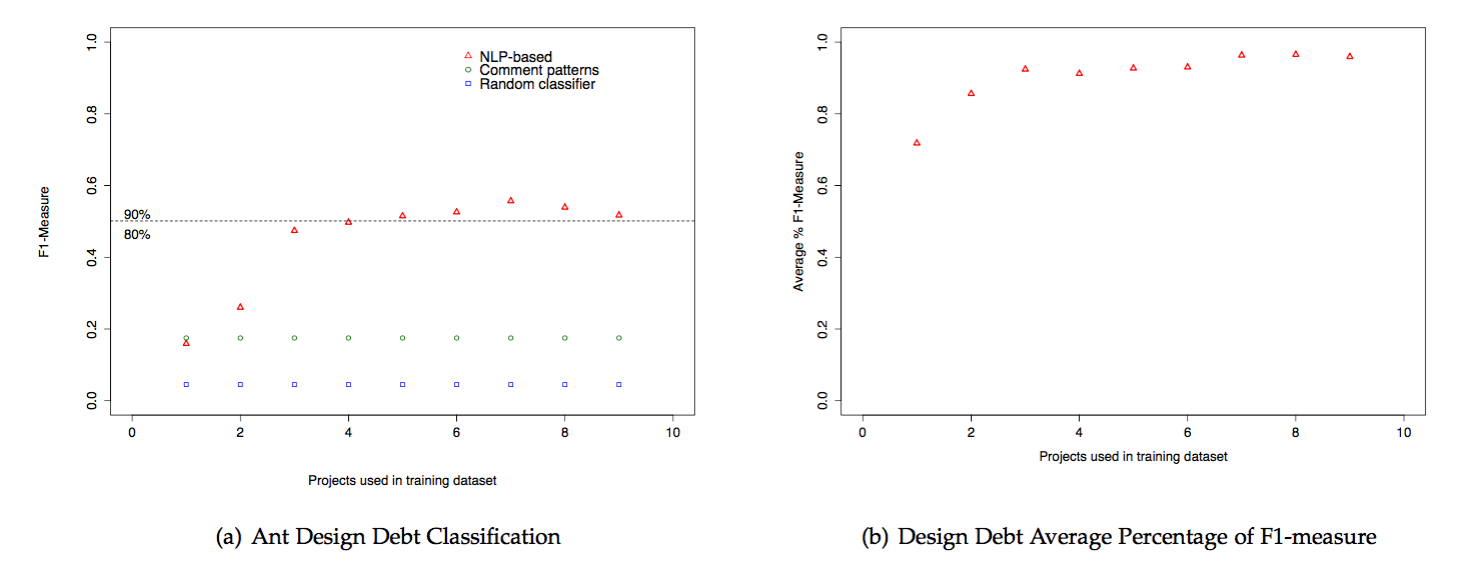
-------------------------------------------------------------------------------------------------------------------

• (R3-9) - I don’t think the appendix on p. 16 is needed — I think this data can & should be integrated into two figures (one for each kind of debt), by removing the training data impact for comment patterns & the random classifier. The authors’ NLP-based approach is clearly superior, so I don’t think it’s useful to learn about the impact of training a random classifier. To achieve just 2 graphs, the y-axis would need to be changed to using the % of the max F measure of each iteration rather than the raw F measure (or something similar that is appropriate for all the graphs on the following pages). I think this data is critically important to support RQ3, and should not be relegated to an appendix.

Thank you for the comment. Indeed, being able to condense the information available in our appendix into two graphs would improve the readability of the paper and also provide the necessary support to our findings in RQ3. As the reviewer points out, just considering the F1-measure achieved by each one of the projects would result in a difficult graph to read, making our argumentation more difficult to understand. We do agree that a good way to merge this information is by using the % of the max F1-measure. Therefore, we modified/added the following text into the manuscript in the RQ3 to address this comment.

“Figure 3(a) shows the F1-measure using different sizes of training data for the Ant project. Due to space, we discuss the results for a representative project (Ant) in this section, however, Figure 3(b) presents the maximum percentage average achieved for all projects in each one of the analyzed iterations.. From Figure 3(a), we find that the maximum F1-measure improves as we increase the number of projects (i.e., iterations), achieving the highest F1-measure in the seventh iteration and slightly decreasing afterwards. The horizontal lines in the figure show the 80% and 90% of the highest F1-measure. We can see from Figure 3(a) that with 1,499 comments (i.e., from 3 projects) and 1,815 comments (i.e., from 4 projects), we can achieve 80% and 90% of the highest F1-measure, respectively. This amounts to a reduction of 37.6% and 24.5% in training data to achieve 80% and 90% of the maximum F1-measure, respectively. Considering the tradeoff in accuracy versus the amount of training data, for Ant, using only 3 or 4 projects provides the best tradeoff.

We analyzed all iterations from all projects to determine the iterations that achieve the best F1-measure performance. To measure that, we calculate the average percentage of the maximum F1-measure for each iteration as shown in Figure 3(b). For example, we take the average percentage of the maximum F1-measure achieved during the first iteration for all projects, then we calculate the same value for all second iterations and so on. We find that, the best performance is achieved during the eighth iteration, with an average maximum F1-measure of 96.57% using (on average) 2,353 comments to create the training dataset. In comparison, the ninth iteration has an average maximum F1-measure of 95.99%, which is slightly lower than the average obtained in the eighth iteration, and uses more comments in the training dataset (i.e., 2,432).”



“Table 6 shows the average percentage of the maximum F1-measure for each iteration. Unlike the case of design debt, for requirement debt, the best F1-measure is achieved in the first iteration. This shows that using as few as 380 comments, we can effectively detect requirement self-admitted technical debt as we can also see from Figure 3(d).”

-------------------------------------------------------------------------------------------------------------------

Sure

-------------------------------------------------------------------------------------------------------------------

Typos/grammar: please see attached pdf. The paper could use a careful reading for typos & grammar.

-------------------------------------------------------------------------------------------------------------------

Easy fixes

-------------------------------------------------------------------------------------------------------------------