An Experience Report on Technical Debt in Pull Requests: Challenges and Lessons Learned

Shubhashis Karmakar
University of Saskatchewan
shk106@usask.ca

Zadia Codabux
University of Saskatchewan
zadiacodabux@ieee.org

Melina Vidoni
Australian National University
melina.vidoni@anu.edu.au

ABSTRACT

GitHub is a collaborative platform for global software development, where Pull Requests (PRs) are essential to bridge code changes with version control. However, developers often trade software quality for faster implementation, incurring Technical Debt (TD). When developers undertake reviewers' roles and evaluate PRs, they can often detect TD instances, leading to either PR rejection or discussions. We investigated whether Pull Request Comments (PRCs) indicate TD by assessing three large-scale repositories: Spark, Kafka, and React. We combined manual classification with automated detection using machine learning and deep learning models. We classified two datasets and found that 37.7 and 38.7% of PRCs indicate TD, respectively. Our best model achieved $F1 = 0.85$ when classifying TD during the validation phase. However, we faced several challenges during this process, which may hint that TD in PRCs is discussed differently from other software artifacts (e.g., code comments, commits, issues, or discussion forums). Thus, we present challenges and lessons learned to assist researchers in pursuing this area of research.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; • **Computing methodologies → Machine learning approaches**.

KEYWORDS

Technical Debt, Pull Request Comments, Mining Software Repositories

ACM Reference Format:
Shubhashis Karmakar, Zadia Codabux, and Melina Vidoni. 2022. An Experience Report on Technical Debt in Pull Requests: Challenges and Lessons Learned. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '22), September 19–23, 2022, Helsinki, Finland*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3544902.3546637

1 INTRODUCTION

Collaborative software development through version control systems such as GitHub leverages Pull Requests (PRs) to support interaction between developers in code repositories [29]. In distributed software development, pull-based development models are one of the most popular contribution models [15], offering a low barrier of entry for potential contributors. PR combines source code changes with discussions in the form of Pull Request Comments (PRCs). Through PRCs, developers indicate style or design conflicts regarding the *proposed* changes [18, 39].

Quite often, developers incur Technical Debt (TD) – shortcuts, and workarounds to minimize the effort required to propose a solution because of a lack of time or knowledge [22]. Prior studies demonstrated that TD is 'admitted' (namely, 'confessed' in natural language); this is known as SATD (Self-Admitted Technical Debt). SATD has been found in several software artifacts, including (but not limited to) source code comments, issue trackers, commit messages, and code reviews [20, 24, 30].

While static code analysis is the most common way of detecting TD, existing static analyzers, based on different rules to extract TD, often have different outcomes [2]. The detection of inadvertent TD [35], accrued unintentionally due to ignorance and oversight, cannot be detected using traditional software artifacts (e.g., source code). Given PRs' relevance to the software development lifecycle and the rich information based on discussions between developers through PRCs [29], we investigated the presence of TD in PRCs. Due to the existence of a large number of PRCs in a code repository, a manual study would not be feasible to uncover all instances of TD. Therefore, we also conducted automation using Machine Learning (ML) and Deep Learning (DL) algorithms, comparing the state-of-the-art techniques to those used in prior studies [3, 10, 11]. We also assessed whether PRC can be an additional source of TD in non-code artifacts. To the best of our knowledge, PRCs have not yet been investigated as a source of TD.

We successfully achieved a binary classification of TD and non-TD in PRCs, both manual and automated, but we uncovered several challenges and thus, proposed mitigation strategies to consider when using PRCs as a TD source. We outlined our process and findings and discussed challenges and lessons learned regarding TD discussion in PRCs. We also released a **replication package**[1] with all the datasets.

This paper is organized as follows. Section 2 describes the related work. Section 3 elaborates on the data collection process. Section 4 reports the TD detection process and results. Section 5 discusses the challenges and lessons learned. Section 6 outlines the threats to validity and Section 7 concludes this work.

2 RELATED WORK

Several studies identified TD through source code inspection [7, 38]. Gat and Heintz [13] analyzed TD using unit testing,

[1] https://doi.org/10.5281/zenodo.6829274

ESEM '22, September 18–23, 2022, Helsinki, Finland

ACM ISBN 978-1-4503-9427-7/22/09…$15.00
https://doi.org/10.1145/3544902.3546637

code coverage, rule conformance, code complexity, duplication of code, documentation, and design characteristics. Das et al. [9] investigated TD through architectural degradation and code smells, and examined different source code TD detection tools.

Additionally, other sources based on natural language have been used. For example, source code comments were the original domain for SATD investigation [30]. Afterward, several studies have been performed both fully manually [24] and automatically using several techniques (including NLP, and sentiment analysis, among others) [11, 25] for classification in many domains. Code reviews with SATD were found to be less likely accepted [20], and SATD was also present when developers depend on updated functionalities to complete their work [23]. Other sources of SATD were issue trackers [4] and commit messages [31].

Although PRs have not yet been approached for TD detection, they have been used for other purposes, e.g., to generate descriptions of proposed changes [10], detect bots' participation in PRs [14], and predict whether some changes will be merged or discarded [18]. A complementary study investigated regex-related bugs in PRs to classify their nature [37]. Given the presence of bugs, other authors analyzed the reliability of tests in updating dependency and suggested the use of adequate tests for all usages of library dependencies so that unintended functionalities are not introduced over time [16]. Finally, considering large-scale ecosystems, Businge et al. [8] analyzed divergent forks of Android, .NET, and JavaScript to uncover maintenance and reuse practices.

Despite TD being widely approached, its investigation in PRs remains a gap in the literature and a potential complementary source of TD. Based on the structure and usage of PRs [29], there is a possibility of missing inadvertent TD, i.e., TD unknowingly introduced or admitted [28].

3 DATA COLLECTION AND PROCESSING

Since this was an exploratory study, we selected active, multipurpose software repositories based on the recommendations of Kalliamvakou et al. [19] to ensure that our data was up-to-date. We also constrained projects with a large number of PRs (open or closed) to have enough data to enable useful insights to be drawn from this study. Based on prior studies [3], we selected three repositories: Apache Spark, Apache Kafka, and React. Table 1 summarizes PRs statistics for each project.

We used multiple queries to extract the PRCs and leveraged GraphQL[2] (an alternative REST-based application to GitHub's API) to minimize them. This did not threaten the validity of our study [6]. Using the GraphQL API[3] provided by GitHub, we extracted PRCs details, including number, comment type, HTML structure, comment timestamp, author, and each comment's author details (including the handle).

In addition, we removed bot PRCs from both open and closed PRs as depicted in Figure 1. As per Golzadeh et al. [14], bots are often identified by their handles (i.e., GitHub username), e.g., for Apache Spark, the bots were `AmplabJenkins`, `SparkQA`, `asfgit`, and `asfbot`. Table 1 summarizes the statistics before ('PRCs') and after ('Filtered Comments') filtering the bot PRCs.

[2]https://graphql.org/
[3]https://docs.github.com/en/graphql

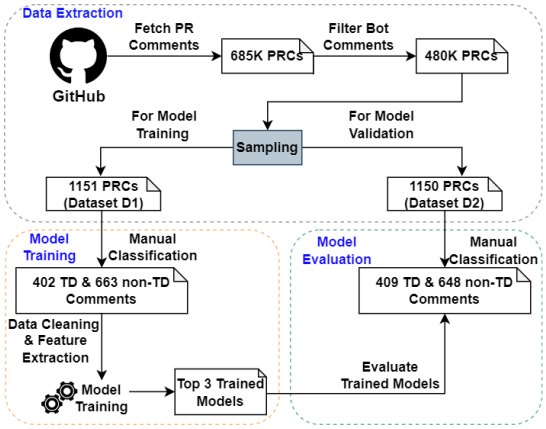

Figure 1: Schematic Diagram of the Study

Table 1: Statistics for the Selected Projects

	Apache Spark	Apache Kafka	React
Total PRs	34,401	11,445	11,542
Open PRs	223	951	235
Closed PRs	34,178	10,494	11,307
PRCs	685,206	160,974	68,185
Filtered Comments	480,750	143,986	58,869

4 TECHNICAL DEBT DETECTION

4.1 Manual Classification

To conduct automated TD detection through supervised approaches, we required a dataset of manually classified PRCs as training data. Thus, we randomly sampled PRCs of Apache Spark, Apache Kafka, and React, using a confidence level of 95% and a margin of error of 5%. This resulted in 384 PRCs each for Spark and Kafka, and 383 for React, resulting in a combined manual sample of **1151 PRCs**. Henceforth, this is referred to as *D*1 across the three systems. We followed the classification of TD types from the ontology provided by Alves et al. [1] and classified our PRCs as TD (those pertaining to a TD type) and non-TD.

All authors classified the PRCs of *D*1 independently. Next, the individual classifications were merged, and disagreements were discussed to obtain the final classification for use as training data. A PRC was removed from *D*1 when the discussion could not resolve a disagreement. We obtained a high agreement rate, measured using Cohen's Kappa coefficient [26] (see Table 2).

Table 2: Manual Identification Summary of *D*1

Repository	Kafka	React	Spark	Total
Total Sampled PRCs	349	359	357	1065
# TD PRCs	137	126	139	402
# Non-TD PRCs	212	233	218	663
TD (%)	39.2	35.1	39.0	37.7
Inter-Rater Agreement	0.87	0.91	0.90	

We noticed that the TD PRCs could further be classified on how developers are discussing the source of the TD. Developers *explicitly* indicate an underlying issue using pre-existing context (e.g., the use of 'hack' or 'crash' in the PRCs), or *implicitly* indicate subtle issues without any former definitions (e.g., as a result of an unintentional design decision) [36]. One PRC could have multiple instances of TD and non-TD. Therefore, we further split 811 TD PRCs (combined from dataset $D1$ and $D2$) into 1286 based on the context. One author split the PRCs, and another verified it; adjustments were made whenever needed, following discussions. Two authors then independently classified the split comments into the categories: *implicit, explicit, and non-TD* and later disagreements were discussed. Cohen's Kappa coefficient was 0.43, which is considered moderate.

4.2 Automated Classification

We pre-processed the PRCs of $D1$ following standard processing steps for Natural Language Processing (NLP) [3], including lowercasing, removing punctuation, mentions, numbers, emails, URLs, and stopwords. Before training the ML/DL models to classify text-based data, the words must be converted into tokens. As our baseline, we convert words to a matrix of token counts, and TF-IDF counts for ML models. Additionally, we used WordPiece [34] and Byte-Pair Encoding (BPE) [12] with the state-of-the-art transformer models (DistilBERT [33], AlBERT [21], RoBERTa [40]). As a baseline, we selected Multinomial Naive Bayes (MNB), Support Vector Machine (SVM), and Maximum Entropy (ME). These ML techniques have been successfully used in text-based classification [17]. Since DL techniques have shown better results for NLP-based tasks [27], we selected Convolutional Neural Network (CNN), Long Short Term Memory (LSTM), and pre-trained Transformer models (DistilBERT, AlBERT, RoBERTa). Because TD identification is a sentence classification task, we used metrics of precision, recall, and F_1 to evaluate the model performance.

Table 3: Model Performance on $D1$

Model	Feature Extractor	Precision	Recall	F1
DistilBERT	Tokenizer	0.76	0.80	0.78
AlBERT	Tokenizer	0.70	0.88	0.78
RoBERTa	Tokenizer	0.81	0.79	0.80
CNN	Tokenizer	0.65	0.73	0.69
LSTM	Tokenizer	0.64	0.74	0.69
MNB	Count Vectorizer	0.67	0.74	0.70
MNB	TF-IDF Vectorizer	0.75	0.32	0.45
SVM (RBF)	Count Vectorizer	0.65	0.73	0.69
SVM (RBF)	TF-IDF Vectorizer	0.73	0.30	0.43
MaxEnt	Count Vectorizer	0.77	0.53	0.62
MaxEnt	TF-IDF Vectorizer	0.75	0.43	0.55

All models were trained using $D1$. We combined layers, learning rates for CNN and LSTM, and hyperparameters for ML models (e.g., alpha for MNB, regularization parameter for SVM and ME). Because the pre-trained Transformer Models (PTMs) are already trained, we fine-tuned these models with different learning weights for TD/non-TD PRCs. $D1$ was imbalanced, with more non-TD than TD. As PTMs are better at sentence classification, we handled the imbalance by penalizing an incorrect TD classification. This was achieved using Loss Function $TD_{weight} = 4 * non - TD_{weight}$; namely, the sample weight was four times the weight of non-TD samples. Using the same weight for TD and non-TD PRCs would have caused a misclassification of TD. Thus, we tried different weight values (e.g., 2–5). When the TD weight is four times, it provides a balanced score for the PTM. We found that CountVectorizer (converting PRCs to a matrix of token counts) is more effective than matrices of TF-IDF features. For TD detection, baseline DLs provided a comparable score to MLs. However, aligned with prior findings related to transformer models [27], the PTMs showed comparatively better performance in sentence classification. Table 3 summarizes the results per model.

To verify the effectiveness of our top-performing models (RoBERTa, AlBERT, and DistilBERT) in TD detection, we randomly sampled another dataset from the PRCs of Apache Spark, Apache Kafka, and React using 95% of confidence and 5% margin of error; this resulted in 384 PRCs each for Spark and Kafka and 382 for React, adding up to **1150**, henceforth, referred to as $D2$). After further cleaning up the PRCs (e.g., remove empty PRCs), we had 1132 PRCs left. All authors manually classified these PRCs independently and without knowing the algorithms' prediction. After the manual classification, the authors discussed the differences and calculated a new inter-rater agreement (summarized in Table 4) using Cohen's Kappa coefficient.

Table 4: Manual Identification Summary of $D2$

Repository	Kafka	Spark	React	Total
Total Sampled PRCs	355	354	348	1057
# TD PRC	140	117	152	409
# Non-TD PRC	215	237	196	648
TD (%)	39.4	33.0	43.6	38.7
Inter-Rater Agreement	0.89	0.90	0.91	

Table 5: Model Performance on $D2$

Model	Feature Extractor	Precision	Recall	F1
DistilBERT	Tokenizer	0.76	0.91	0.82
AlBERT	Tokenizer	0.70	0.95	0.80
RoBERTa	Tokenizer	0.82	0.87	0.85

Then, we evaluated the performance of RoBERTa, AlBERT, and DistilBERT with $D2$. We summarized the precision, recall and F1-score for these models in Table 5. We obtained similar scores for the PTMs as depicted in Tables 3 and 5, confirming PTMs generalization capability. The inter-rater agreement of our manual classification is comparable, indicating a high agreement (Tables 2, 4).

Although our results seem promising (high precision, recall, and $F1$) and we successfully identified TD instances on PRCs, we faced several issues. Section 5 elaborates on the challenges encountered, presenting some mitigation strategies.

5 CHALLENGES AND LESSONS LEARNED

This section discusses the challenges we faced while identifying TD in PRCs, including examples and possible mitigation strategies. The replication package[1] includes more examples for each challenge.

Challenge 1. PRs are not self-contained; information from other artifacts will affect whether a PRC implies TD or not.
Lesson Learned 1. PRs information is insufficient and should be triangulated with other sources (e.g., interviews, observations) and analyzed alongside other artifacts (e.g., associated source code).

Description. Pull requests are inherently variable, as they are directly tied into a software system's domain and constraints which may not be publicly available (e.g., people's skills, restricted budgets, and time-frames). Prior works demonstrated that elements such as styling could affect a PR's merge [41] and the sentiment conveyed through wording can also impact the PR's process [29]. We found that it is typical for a PR's discussion to continue face to face, limiting the available information: `We talked more offline about this and discovered that [...]`. This adds further challenges to the identification because that information can only be accessed through interviews/observations; it may be possible that the aforementioned 'offline' talk refers to other tools (e.g., Slack), but accessing those are sometimes not possible and may violate organizational policies.

Also, there are 'unspoken rules' regarding handling specific changes that may lead to postponing an item (as per Challenge 2). These rules may not be publicly specified (e.g., `We don't normally merge things that are subjective, because it would lead to a lot of PRs.`), or ongoing managerial discussions may affect the entire infrastructure (e.g., `There are some plans to run the Kafka system tests with other clients, and that would hopefully show issues like this`). Such comments demonstrate that PRCs may not provide enough context to understand whether a statement should be considered TD, posing a threat to the study's validity.

Mitigation. Because PRCs are not enough on their own, triangulating with additional data sources will help provide more context and better understand the 'conversation' around the changes; this includes source code analyses, including parsing of source code comments, but can be extended to process-related data such as interviews or observations.

Challenge 2. A PRC may have the potential to become a TD instance, but whether it becomes TD depends on decisions that have not yet been taken or are not explicitly mentioned in the PRC.
Lesson Learned 2. The developers' processes, including their management of PRs, will affect Potential TD.

Discussion. Li et al. [22] defined TD as "technical compromises that can yield short-term benefits but may hurt the long-term health of a system", while Martin Fowler conceptualized the four quadrants of awareness for TD introduction[4], determining that TD may be unwillingly introduced. TD has been explored mostly in source code comments [24, 25, 30] and commit messages [31], where the situations leading to the introduction, repayment, or prevention of TD already occurred. However, PRs are meant to provide code-review capabilities and may or may not result in the acceptance of a code change into the main code because of several factors (e.g., code style or inconsistencies with other PRs)[41].

Thus, a PRC may indicate *potential TD*–i.e., it is not an immediate issue but has the potential to become an issue [32] if the developers make certain decisions and their subsequent actions result in TD. In some cases, it may be possible to triangulate the PRCs with other data sources (e.g., code changes) to understand the context better. However, this may not always be possible as it could still be an ongoing discussion. For example, `...we can now consider implementing a much more targeted pooling strategies for events` is a case of potential TD because implementing a specific solution is possible and reasonable, but not doing so will incur TD; however, adding such a feature can be relegated to another PR as per the development process.

The above example leads to Lesson 3. Although there are guidelines on how PRs are used, each project has its nuances [41], which limits how closely related changes must be to be part of the same PR. Therefore, it is possible that a seemingly postponed change is only 'redirected' into another PR for the sake of the process. For example, a contributor commented: `For line 376 above, which is out of scope of this PR: [...]`, and the decision to move and address it in another PR, creates *potential TD*.

Mitigation. The concept of *potential TD* has been discussed but not formally defined; to our knowledge, it has not been explicitly investigated either. Understanding potential TD, the transition between states (from potential to real or avoided), and when to consider it will allow a thorough analysis of TD in PRCs.

Challenge 3. Unlike other sources of TD, PRCs can be ambiguous. This results in TD taking different shapes and showcasing different degrees of explicitness.
Lesson Learned 3. TD in PRCs is not enough to determine the debt instances of software due to the explicitness of comments, developers' attitudes, and nature of the discussions.

Discussion. Because PRCs are meant to discuss ongoing changes, some TD instances are *explicit* and highlight an existent, specific issue using words not subject to interpretation. This is the 'traditional' wording of TD-admission seen in source code comments [24, 25, 30], which can be linked to specific TD management activities [22]. For example `FWIW this does seem to have fixed the error I saw on Facebook. (Just repro'ed and no longer breaks.)` explicitly indicates repayment, while `I still see the following issue locally: [...]` indicates an existent case of Defect Debt.

However, based on the versatility of natural language and the discussion-orientation of PRs, other TD instances are *implicit*–they do not use keywords (e.g., TODO) and cannot/should not be analyzed word by word. For example, `Edit: If I disable the "minify" preset when compiling, the original Line is not null (but it is also incorrect).` indicates Defect Debt without using common keywords. Question-style wording is also used in *implicit* cases, possibly indicating developers are

[4] https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

not confident; e.g., `Since stop closes and sets consumer to null, perhaps we dont need this here?`

Developers participate differently in PRs. For instance, some are insecure, new to a project, or simply making a voluntary contribution [29] and this, in turn, affects whether a PR is merged or not. As a result, the developers' 'affective' state or attitudes will affect the wording used, thus causing automated TD detection to be more challenging.

Mitigation. For PRCs, it may not be enough to classify comments (or sentences, as per Challenge 4) into the common TD vs. non-TD, given that the manifestation of TD is more nuanced, and disregarding the subjective wording can make automatic detection harder. As part of our replication package[1], we provide a small dataset of TD PRCs classified into explicit or implicit (ETD or ITD, respectively). This is also related to Challenge 3.

Challenge 4. A single TD instance can be scattered across multiple PRCs. Likewise, each PRC could disclose multiple instances and TD types.
Lesson Learned 4. Sentence splitting is not straightforward, and guiding it by punctuation, sentences, or paragraphs may threaten the study's validity.

Discussion. A recurrent issue of PRCs is their variability. Some PRCs are concise and composed of a single sentence clearly outlining TD, but this is not always the case. Many PRCs have multiple paragraphs and sentences, discussing many TD types at different degrees of explicitness. This poses an issue for automatic detection, as words and sentences that are non-TD will be 'grouped' with TD instances as training data. As a solution, we tried splits by paragraph, sentences, or semantic splitting through existing libraries[5]. However, the paragraphs were still too large and contained multiple TD types. Sentences can be incorrectly split (e.g., punctuation is not always ideal), opposite to the paragraph-split, as a sentence containing TD may only be understood when associated with another sentence; e.g., `But the list-test is a superset of the map test? Seems to be redundant to me.`. The first rhetorical question could be interpreted as 'discussion' or 'thought exposition' if read independently, but it indicates TD alongside the following sentence. The existent libraries we tested did not provide a good enough result for splitting.

Mitigation. TD classification is done in a binary, exclusionary form; namely, each PRC or sentence can be TD or non-TD. Although manually generated gold data can provide an acceptable degree of sentence separation (as per the dataset of Challenge 3), an automated separation may require specific training. A less resource-demanding approach for automated classification could provide a 'category belonging' akin to what LDA (Latent Dirichlet Association) does with topic modeling [5]. For example, the classification result for a paragraph could be provided as proportions, in the shape of `[NTD=0.1, ETD=0.4, ITD=0.5]`, which would be read as having 50% possibility of containing implicit TD, 40% explicit TD, and 10% non-TD.

[5]https://github.com/bminixhofer/nnsplit

6 THREATS TO VALIDITY

The authors performed independent coding and discussed disagreements to reduce bias for the TD vs. non-TD identification and threats associated with the unfamiliarity of the domain knowledge for the selected systems. This also helped alleviate the threats related to the PRCs lacking context for classification. To find the best model for detecting TD from PRCs, we compared previously used algorithms [27]. We re-sampled the data and manually verified it to generalize the models' performance. The selected systems used different programming languages from diverse domains that were used in previous studies [20] and followed existing guidelines to minimize threats and ensure diversity [19]. Therefore, the challenges and the reported lessons learned are solely based on the PRs we analyzed and might not apply to other projects or domains.

7 CONCLUSION

We investigated TD in PRCs for three open-source projects (Apache Spark, Apache Kafka, and React). We mined and classified PRCs into TD/non-TD. First, we sampled the dataset, manually classified the PRCs, and then automated the process through ML, DL, and PTMs. The PTMs had higher performance overall compared to the ML and DL models. Lastly, we classified the TD PRCs as explicit or implicit TD.

Although we successfully classified PRCs as TD/non-TD and the TD PRCs as explicit/implicit, the classification process was challenging. We uncovered that PRCs often lack the context to understand and correctly classify it, and some were potential TD, but we could not determine whether they will eventually become a TD instance using only information in the PRC. PRCs were often ambiguous and hard to understand due to the natural language in which they were written, and the terminologies and vocabulary used differed among the developers. Lastly, more than one PRC could be related to the same debt instance, or multiple debt instances may be discussed in a PRC. We discussed some mitigation strategies for these challenges.

PRC as a source of TD opens up further research avenues. Analyzing the other software artifacts (e.g., source code, issues, and commit messages) impacted by the PR and conducting ethnographic studies with developers will provide more context and a better understanding of the PRCs. More investigation is also needed to understand how to automatically split the PRCs to be self-contained and contain distinct instances of debt. Using topic modeling to identify the areas of discussion in the PRCs will also help classify the PRCs according to TD types (e.g., code, test, and documentation debt). Identifying the most common TD types in PRCs and determining the characteristics of TD from PRCs concerning the size of TD will also help provide more insights into TD in PR.

ACKNOWLEDGMENTS

This study is partly supported by the Natural Sciences and Engineering Research Council of Canada, RGPIN-2021-04232 and DGECR-2021-00283 at the University of Saskatchewan.

REFERENCES

[1] Nicolli S.R. Alves, Leilane F. Ribeiro, Vivyane Caires, Thiago S. Mendes, and Rodrigo O. Spínola. 2014. Towards an Ontology of Terms on Technical Debt. In *Int. Workshop on Managing Technical Debt*. IEEE, Victoria, BC, Canada, 1–7. https://doi.org/10.1109/MTD.2014.9

[2] Paris C. Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Francesca Arcelli Fontana, Terese Besker, Alexander Chatzigeorgiou, Valentina Lenarduzzi, Antonio Martini, Athanasia Moschou, Ilaria Pigazzini, Nyyti Saarimaki, Darius Daniel Sas, Saulo Soares de Toledo, and Angeliki Agathi Tsintzira. 2021. An Overview and Comparison of Technical Debt Measurement Tools. *IEEE* 38, 3 (2021), 61–71. https://doi.org/10.1109/MS.2020.3024958

[3] Gabriele Bavota and Barbara Russo. 2016. A Large-Scale Empirical Study on Self-Admitted Technical Debt. In *Conf. on Mining Software Repositories*. IEEE, Austin, TX, USA, 315–326.

[4] Stephany Bellomo, Robert L. Nord, Ipek Ozkaya, and Mary Popeck. 2016. Got Technical Debt? Surfacing Elusive Technical Debt in Issue Trackers. In *Conf. on Mining Software Repositories*. IEEE, Austin, TX, USA, 327–338.

[5] David M Blei and Michael I Jordan. 2003. Modeling Annotated Data. In *Conf. on Research and Development in Information Retrieval*. Association for Computing Machinery, New York, NY, USA, 127–134.

[6] Gleison Brito and Marco Tulio Valente. 2020. REST vs GraphQL: A Controlled Experiment. In *Int. Conf. on Software Architecture*. IEEE, Salvador, Brazil, 81–91. https://doi.org/10.1109/ICSA47634.2020.00016

[7] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka. 2010. Managing Technical Debt in Software-Reliant Systems. In *Workshop on Future of SE Research (FoSER '10)*. Association for Computing Machinery, New York, NY, USA, 47–52. https://doi.org/10.1145/1882362.1882373

[8] John Businge, Moses Openja, Sarah Nadi, and Thorsten Berger. 2022. Reuse and Maintenance Practices Among Divergent Forks in Three Software Ecosystems. *Empirical SE* 27, 2 (2022), 1–47.

[9] Dipta Das, Abdullah Al Maruf, Rofiqul Islam, Noah Lambaria, Samuel Kim, Amr S Abdelfattah, Tomas Cerny, Karel Frajtak, Miroslav Bures, and Pavel Tisnovsky. 2022. Technical Debt Resulting from Architectural Degradation and Code Smells: A Systematic Mapping Study. *Applied Computing Review* 21, 4 (2022), 20–36.

[10] Sen Fang, Tao Zhang, You-Shuai Tan, Zhou Xu, Zhi-Xin Yuan, and Ling-Ze Meng. 2022. PRHAN: Automated Pull Request Description Generation Based on Hybrid Attention Network. *Journal of Systems and Software* 185 (2022), 111160.

[11] Gianmarco Fucci, Nathan Cassee, Fiorella Zampetti, Nicole Novielli, Alexander Serebrenik, and Massimiliano Di Penta. 2021. Waiting Around or Job Half-Done? Sentiment in Self-Admitted Technical Debt. In *Int. Conf. on Mining Software Repositories*. Springer, Madrid, Spain, 403–414. https://doi.org/10.1109/MSR52588.2021.00052

[12] Philip Gage. 1994. A New Algorithm for Data Compression. *C Users J.* 12, 2 (1994), 23–38.

[13] Israel Gat and John D. Heintz. 2011. From Assessment to Reduction: How Cutter Consortium Helps Rein in Millions of Dollars in Technical Debt. In *Int. Workshop on Managing Technical Debt*. Association for Computing Machinery, New York, NY, USA, 24–26. https://doi.org/10.1145/1985362.1985368

[14] Mehdi Golzadeh, Alexandre Decan, Damien Legay, and Tom Mens. 2021. A Ground-Truth Dataset and Classification Model for Detecting Bots in GitHub Issue and PR Comments. *Journal of Systems and Software* 175 (2021), 110911. https://doi.org/10.1016/j.jss.2021.110911

[15] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. Work practices and challenges in pull-based development: The integrator's perspective. In *ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, Florence, Italy, 358–368.

[16] Joseph Hejderup and Georgios Gousios. 2022. Can We Trust Tests to Automate Dependency Updates? A Case Study of Java Projects. *Journal of Systems and Software* 183 (2022), 111097.

[17] M Ikonomakis, Sotiris Kotsiantis, and V Tampakas. 2005. Text Classification Using Machine Learning Techniques. *Trans. on Computers* 4, 8 (2005), 966–974.

[18] Khairul Islam, Toufique Ahmed, Rifat Shahriyar, Anindya Iqbal, and Gias Uddin. 2022. Early Prediction for Merged vs Abandoned Code Changes in Modern Code Reviews. *Information and Software Technology* 142 (2022), 106756. https://doi.org/10.1016/j.infsof.2021.106756

[19] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Conf. on Mining Software Repositories*. Association for Computing Machinery, New York, NY, USA, 92–101. https://doi.org/10.1145/2597073.2597074

[20] Yutaro Kashiwa, Ryoma Nishikawa, Yasutaka Kamei, Masanari Kondo, Emad Shihab, Ryosuke Sato, and Naoyasu Ubayashi. 2022. An Empirical Study on Self-Admitted Technical Debt in Modern Code Review. *Information and Software Technology* 146 (2022), 106855. https://doi.org/10.1016/j.infsof.2022.106855

[21] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations.. In *ICLR*. OpenReview.net, Addis Ababa, Ethiopia.

[22] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A Systematic Mapping Study on Technical Debt and its Management. *Journal of Systems and Software* 101 (2015), 193–220. https://doi.org/10.1016/j.jss.2014.12.027

[23] Rungroj Maipradit, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2020. Wait for It: Identifying "On-Hold" Self-Admitted Technical Debt. *Empirical SE* 25, 5 (2020), 3770–3798. https://doi.org/10.1007/s10664-020-09854-3

[24] Everton da S. Maldonado and Emad Shihab. 2015. Detecting and Quantifying Different Types of Self-Admitted Technical Debt. In *Int. Workshop on Managing Technical Debt*. IEEE, Bremen, Germany, 9–15. https://doi.org/10.1109/MTD.2015.7332619

[25] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt. *Transactions on SE* 43, 11 (2017), 1044–1062. https://doi.org/10.1109/TSE.2017.2654244

[26] Mary McHugh. 2012. Interrater Reliability: the Kappa Statistic. *Biochemia medica* 22 (10 2012), 276–82. https://doi.org/10.11613/BM.2012.031

[27] Shervin Minaee, Nal Kalchbrenner, Erik Cambria, Narjes Nikzad, Meysam Chenaghlu, and Jianfeng Gao. 2021. Deep Learning–Based Text Classification: A Comprehensive Review. *ACM Comput. Surv.* 54, 3, Article 62 (2021), 40 pages. https://doi.org/10.1145/3439726

[28] Arthur-Jozsef Molnar and Simona Motogna. 2020. Long-Term Evaluation of Technical Debt in Open-Source Software. In *Int. Symp. on Empirical SE and Measurement*. Association for Computing Machinery, New York, NY, USA, Article 13, 9 pages. https://doi.org/10.1145/3382494.3410673

[29] Marco Ortu, Giuseppe Destefanis, Daniel Graziotin, Michele Marchesi, and Roberto Tonelli. 2020. How do you Propose Your Code Changes? Empirical Analysis of Affect Metrics of Pull Requests on GitHub. *Access* 8 (2020), 110897–110907. https://doi.org/10.1109/ACCESS.2020.3002663

[30] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *Int. Conf. on Software Maintenance and Evolution*. IEEE, Victoria, BC, Canada, 91–100. https://doi.org/10.1109/ICSME.2014.31

[31] Leevi Rantala and Mika Mäntylä. 2020. Predicting Technical Debt from Commit Contents: Reproduction and Extension with Automated Feature Selection. *Soft. Quality Journal* 28, 4 (2020), 1551–1579. https://doi.org/10.1007/s11219-020-09520-3

[32] Gabriela Robiolo, Ezequiel Scott, Santiago Matalonga, and Michael Felderer. 2019. Technical Debt and Waste in Non-functional Requirements Documentation: An Exploratory Study. In *Product-Focused Software Process Improvement*, Xavier Franch, Tomi Männistö, and Silverio Martínez-Fernández (Eds.). Springer Int. Publishing, Cham, 220–235.

[33] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, A Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. *CoRR* abs/1910.01108 (2019).

[34] Mike Schuster and Kaisuke Nakajima. 2012. Japanese and Korean voice search. In *Int. Conf. on Acoustics, Speech and Signal Processing*. IEEE, Kyoto, Japan, 5149–5152. https://doi.org/10.1109/ICASSP.2012.6289079

[35] Edith Tom, Aybüke Aurum, and Richard Vidgen. 2013. An exploration of technical debt. *Journal of Systems and Software* 86, 6 (2013), 1498–1516. https://doi.org/10.1016/j.jss.2012.12.052

[36] Johannes Holvitie Tomi'bgt'Suovuo, Jouni Smed, and Ville Leppänen. 2015. Mining Knowledge on Technical Debt Propagation. *14th Symposium on Programming Languages and Software Tools* 1525 (2015), 281–295.

[37] Peipei Wang, Chris Brown, Jamie A Jennings, and Kathryn T Stolee. 2022. Demystifying Regular Expression Bugs. *Empirical SE* 27, 1 (2022), 1–35.

[38] Nico Zazworka, Antonio Vetro', Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman, and Forrest Shull. 2014. Comparing Four Approaches for Technical Debt Identification. *Software Quality Journal* 22, 3 (sep 2014), 403–426.

[39] Xin Zhang, Yang Chen, Yongfeng Gu, Weiqin Zou, Xiaoyuan Xie, Xiangyang Jia, and Jifeng Xuan. 2018. How do Multiple Pull Requests Change the Same Code: A Study of Competing Pull Requests in GitHub. In *IEEE Int. Conf. on Software Maintenance and Evolution*. IEEE, Madrid, Spain, 228–239. https://doi.org/10.1109/ICSME.2018.00032

[40] Liu Zhuang, Lin Wayne, Shi Ya, and Zhao Jun. 2021. A Robustly Optimized BERT Pre-training Approach with Post-training. In *Chinese National Conference on Computational Linguistics*. Chinese Information Processing Society of China, Huhhot, China, 1218–1227.

[41] Weiqin Zou, Jifeng Xuan, Xiaoyuan Xie, Zhenyu Chen, and Baowen Xu. 2019. How Does Code Style Inconsistency Affect Pull Request Integration? An Exploratory Study on 117 GitHub Projects. *Empirical Software Engineering* 24 (12 2019). https://doi.org/10.1007/s10664-019-09720-x