Automated Software Engineering (2022) 29:53
https://doi.org/10.1007/s10515-022-00358-6

Self-admitted technical debt in R: detection and causes

Rishab Sharma[1] · **Ramin Shahbazi**[1] · **Fatemeh H. Fard**[1] · **Zadia Codabux**[2] · **Melina Vidoni**[3]

Received: 16 December 2021 / Accepted: 8 August 2022 / Published online: 25 August 2022

Abstract

Self-Admitted Technical Debt (SATD) is primarily studied in Object-Oriented (OO) languages and traditionally commercial software. However, scientific software coded in dynamically-typed languages such as R differs in paradigm, and the source code comments' semantics are different (i.e., more aligned with algorithms and statistics when compared to traditional software). Additionally, many Software Engineering topics are understudied in scientific software development, with SATD detection remaining a challenge for this domain. This gap adds complexity since prior works determined SATD in scientific software does not adjust to many of the keywords identified for OO SATD, possibly hindering its automated detection. Therefore, we investigated how classification models (traditional machine learning, deep neural networks, and deep neural Pre-Trained Language Models (PTMs)) automatically detect SATD in R packages. This study aims to study the capabilities of these models to classify different TD types in this domain and manually analyze the causes of each in a representative sample. Our results show that PTMs (i.e., RoBERTa) outperform other models and work well when the number of comments labelled as a particular SATD type has low occurrences. We also found that some SATD types are more challenging to detect. We manually identified sixteen causes, including eight new causes detected by our study. The most common cause was *failure to remember*, in agreement with previous studies. These findings will help the R package authors automatically identify SATD in their source code and improve their code quality. In the future, checklists for R developers can also be developed by scientific communities such as rOpenSci to guarantee a higher quality of packages before submission.

Keywords Self-admitted technical debt · R packages · Machine learning · Deep learning · Deep neural pre-trained language models

✉ Melina Vidoni
melina.vidoni@anu.edu.au

Extended author information available on the last page of the article

1 Introduction

Software practitioners strive to implement high-quality software. In both traditional (mostly Object-Oriented (OO) and commercial) and scientific software, they often rush to complete tasks for multiple reasons, such as cost reduction, short deadlines, or even lack of knowledge (da Silva Maldonado et al. 2017). However, the effects of these actions are even more relevant in scientific software since they are used to process research results in many disciplines.

'Scientific' and 'traditional' software have striking differences (Hannay et al. 2009). Scientific software developers are versed in the domain (i.e., the science) and will become the end-users of the software they create (Pinto et al. 2018). However, in 'traditional' software development (commercial applications), developers follow a specific set of requirements. Additionally, scientific software is aimed at understanding a problem rather than obtaining commercial benefits (Pinto et al. 2018; German et al. 2013). The difference between 'traditional' and 'scientific' software is not due to the programming language's age or name but the purpose of the software itself. In particular, "part of the complexity in measuring the scientific software ecosystem comes from how those different pieces of software are brought together and recombined into workflows and assemblies" (Howison et al. 2015). In package-based environments, new research software 'runs off' previous software (namely, packages), and its maintainability and quality can be affected by that of the packages it relies on, thus affecting the validity of research results in other disciplines (Arvanitou et al. 2021). However, multiple issues sprout from package-based environments.

"Software engineering research has traditionally focused on studying the development and evolution processes of individual software projects" (Decan et al. 2016), while the concept of package-based ecosystem goes beyond an isolated software querying and retrieving data from an API (Application Programming Interface). Prior works demonstrated that the openness and scale of package-based environments (such as R's CRAN, Python's PyPi, and Node.js' npm) "lead to the spread of vulnerabilities through package network, making the vulnerability discovery much more difficult, given the heavy dependence on such packages and their potential security problems" (Alfadel et al. 2021). Moreover, frequent package updates (even to fix bugs) often bring breaking changes to the packages that depend on them, and packages with many dependencies eventually become unmaintainable (Mukherjee et al. 2021). More importantly, each package imported also brings further imports (known as 'transitive dependencies'), which "need to be kept updated to prevent vulnerabilities and bug propagation that might endanger the whole ecosystem" (Mora-Cantallops et al. 2020b). Prior research found that this can be controlled by the community values supporting those ecosystems (Bogart et al. 2016). When compared to simple, isolated APIs, package-based ecosystems are so complex that various researchers compared software ecosystems with natural ecosystems as they also grow and evolve (Mora-Cantallops et al. 2020a).

There are many languages and environments for scientific computing. However, recently R has gained ubiquity in studies regarding statistical analysis and

mathematical modelling and has been one of the fastest-growing programming languages (Zanella and Liu 2020). Not only it is package-based (Pinto et al. 2018), but it is also multi-paradigm and with an open community that actively promotes creating open-source packages (Codabux et al. 2021). In May 2021, R ranked 13th in the TIOBE index, which measures the popularity of programming languages, reaching the highest position (8th place) in August 2020 (TIOBE 2020). In 2021, it was ranked as the 7th most popular language by the IEEE Spectrum.[1] Despite being a popular programming language, R developers do not see themselves as 'true programmers' and lack a formal programming education (German et al. 2013; Pinto et al. 2018), hence possibly prioritizing other activities over quality assurance and defect-free code.

Technical Debt (TD) is a metaphor reflecting the implied cost of additional rework caused by choosing an easy solution now instead of a better approach that would take longer (Maldonado and Shihab 2015). *Self-Admitted Technical Debt* (SATD) refers to situations where the developers are aware that the current implementation is not optimal and write comments alerting of the problems (Potdar and Shihab 2014). Through SATD, developers consciously perform a hack and 'record' it by adding comments as a reminder (or as an admission of guilt) (Sierra et al. 2019). Wehaibi et al. (2016) reported that SATD has an impact on software maintenance as SATD changes are more complex than non-TD changes. Most SATD studies to date were conducted in the domain of OO software repositories (Sierra et al. 2019; Potdar and Shihab 2014; da Silva Maldonado et al. 2017; Flisar and Podgorelec 2019). Vidoni (2021b) manually identified SATD comments in R packages but did not provide any tool to detect them automatically. SATD is understudied, especially in scientific software such as R. This contributes to a known research gap already recognized in multiple studies. In particular, as Storer (2017) stated, "the 'gap' or 'chasm' between software engineering (SE) and scientific programming is a serious risk to the production of reliable scientific results." This gap is even more relevant considering that prior work on SATD in R packages demonstrated that R comments differ from Java, consist of different TD types, and are characterized by different keywords (Vidoni 2021b).

To our knowledge, there is no research on automatic detection of SATD in R, which is a language with striking differences compared to OO and commercial software. Thus, this study aims to investigate the capabilities of traditional Machine Learning (ML), deep learning, and deep neural Pre-Trained Language Models (PTMs) for automatic detection of SATD in R packages. We selected techniques that have already been used for automated SATD detection in OO (Ren et al. 2019; Yan et al. 2018; Flisar and Podgorelec 2019; da Silva Maldonado et al. 2017), and compared them to PTMs. The latter was chosen given they have been used to analyze natural language related to software (Zhang et al. 2020) but not applied to SATD. However, they demonstrated excellent capabilities to handle small SE-related datasets (Robbes and Janes 2019). We used a dataset of R source code comments previously classified into 12 TD types (Vidoni 2021b), and automatically classified it

[1] See: https://spectrum.ieee.org/top-programming-languages-2021.

using several techniques to compare the performance among techniques. Additionally, we conducted manual classifications to derive new and complementary data related to SATD characteristics.

Our findings also show that PTMs (RoBERTa model) outperform other models for identifying SATD types, especially when the number of comments for a SATD type is low (meaning, it works well even with limited training data). Note that PTMs have never been used in SATD detection before. Our F1 metric (a weighted average of precision and recall, a metric used to evaluate classifier algorithms) indicates that some types of SATD in R are easier to identify (*Code*, *Test*, *Versioning* and non-SATD comments), while others are not (e.g., *Algorithm* and *People*). We also uncovered eight new causes of SATD introduction, totaling 16. Irrespective of the TD type, *failure to remember* is the most common cause of SATD. *Inconsistent communication* (among developers) and *workarounds or hacks* are also quite common.

Our main contributions are as follows:

- This is the first automated detection analysis of SATD in R programming, specifically for R packages.
- Likewise, PTMs for SATD detection have not been used before.
- An augmented corpus (from 8 to 16) of plausible causes of SATD, extracted from 1,345 comments. It expands on previously proposed categories and is publicly shared.
- The automated detection of 12 types of SATD compared to 5 types in other SATD studies.

Paper Structure. Sect. 2 covers related work and how our study differentiates from the existing literature. Sect. 3 outlines the methodology including our research questions, data processing, experimental setup, and definitions of evaluation metrics. Sect. 4 presents our results. Discussions and implications are presented in Sect. 5 followed by threats to validity in Sect. 6. We conclude our work in Sect. 7.

2 Related work

This section covers different related work organized by areas.

General SATD Potdar and Shihab (2014) manually classified source code comments to obtain 62 SATD patterns. The most common types of TD and the heuristics to discover them were also investigated (Maldonado and Shihab 2015; Potdar and Shihab 2014). These studies were expanded upon by a large-scale automated replication, generating another manually classified dataset, later used in multiple follow up works. Bavota and Russo (2016) and da Silva Maldonado et al. (2017) applied Natural Language Processing (NLP) on that dataset to automatically mine and detect SATD occurrences in ten Open Source Projects (OSPs). They determined that *Code Debt* represents almost 30% of the occurrences and that specific words related to mediocre code are the best indicators of *Design Debt*. Flisar and Podgorelec (2019) created an automated prediction system to estimate SATD in the comments of OO software by comparing different methods (three feature

selection and three text classification). Fucci et al. (2021) did a manual classification of about 1k of Java comments from the sample of (da Silva Maldonado et al. 2017), and worked only on Defect, Design, Documentation and Implementation Debt, to annotate sentiments as negative and non-negative; their results determined most and concluded that comments related to functional problems tend to be negative. Yan et al. (2018) investigated whether software changes introduce SATD by performing an empirical study on OSPs, to assess SATD at the moment it was admitted.

SATD was also assessed in other domains outside source code comments. Li et al. (2020) identified eight TD types in issue trackers, determining that identified TD is mostly repaid, and by those that identified it or created it; however, they focused only on two large-scale Java projects and centred on TD occurrence rather than in its automated detection. Another study focused on the *acquisition and mining* of SATD in issue trackers by pre-labelling issues in issue trackers (Xavier et al. 2020); their goal was not to automate the identification through machine learning, but with process updates, and they only assessed five large-scale Java projects.

Overall, the main difference between these studies and ours is that they focused only on purely Object-Oriented (OO), large-scale projects mainly developed in Java. Moreover, none of them applied PTMs as a detection technique.

Automated SATD detection Detecting instances of SATD has been a manual or automatic process, but we present works on the latter. Ren et al. (2019) proposed a Convolutional Neural Network (CNN) to classify source code comments as SATD and non-SATD, using an existing dataset (Maldonado and Shihab 2015) and extracted more comprehensive and diverse SATD patterns than the manual extraction. Zampetti et al. (2020) used a manual classification of SATD removal dataset (Zampetti et al. 2018) to remove SATD automatically using CNN and Recurrent Neural Network (RNN) and reported that their work outperforms the human baseline. Santos et al. (2020) performed a controlled experiment using a Long Short-Term Memory (LSTM) neural network model combined with Word2vec to detect *Design* and *Requirement* SATD using the existing datasets (Maldonado and Shihab 2015; da Silva Maldonado et al. 2017), and detected that it improved recall and F1 measures. Wattanakriengkrai et al. (2018) focused on *Design* and *Requirements* SATD, using N-gram Inverse Document Frequency and feature selection and experimented with 15 ML classification algorithms using the 'auto-sklearn' for automated SATD detection, and obtained better outcomes in detecting *Design Debt*. Maipradit et al. (2020a) used N-gram feature extraction and 'auto-sklearn' to identify instances of "on-hold" SATD; namely, developers' comments about holding off further implementation work due to factors they cannot control. These authors confirmed their approach as positive to find instances of on-hold SATD, but did not consider TD-types, instead working with a simple binary detection.

Other studies employed plain text mining (Huang et al. 2018; Liu et al. 2018; Mensah et al. 2016). Most used NLP on a manually labeled dataset to automate the SATD detection process. However, Flisar and Podgorelec (2018) used an unlabeled dataset of OSPs and word embeddings for automated SATD detection for a binary (SATD and non-SATD) classification, obtaining about 82% of correct predictions for SATD.

Regarding SATD mined from areas outside source code comments, in a recent work, Rantala and Mäntylä (2020) also worked atop a previously labeled dataset, used logistic Lasso regression to select predictor words in a bag-of-words approach; like all previous papers, they did not consider PTMs and only used the same large-scale Java projects from Maldonado and Shihab (2015). Finally, (AlOmar et al. 2022) provided a tool called SATDBailiff to automated SATD detection, based on a prior plugin (Liu et al. 2018); like before, the model was constructed by reusing the same Java projects reused in multiple studies and did not consider the application of BERT.

R programming Few studies focus on software engineering for R, creating a gap in research. A mining study explored how the use of GitHub influences the R ecosystem regarding the distribution of R packages and for inter-repository package dependencies (Decan et al. 2016). In terms of programming theory, Morandat et al. (2012) assessed the success of different R features to evaluate the fundamental choices behind the language design. These studies focus on dependency management and language design choices for R programming, but not TD. A mixed-methods Mining Software Repositories (MSR) study combined the exploration of GitHub repositories and developers' survey to assess *Test Debt* in R packages (Vidoni 2021a), determining a significant presence of *Test Debt* smells. This semi-automated approach did not apply ML techniques, using only pre-existing testing tools. Codabux et al. (2021) explored the peer-review process of rOpenSci and proposed a taxonomy of TD catered to R packages, determining that reviewers report *Documentation Debt* issues the most. However, it was a manual classification using card sorting. Finally, Vidoni (2021b) used a mixed-methods MSR study and explored 164K comments to determine the existence of SATD in R-packages. It was mostly manual, and its datasets were used as a baseline in this study. However, this dataset generated was not linked to package names, to preserve the identity of the survey participants (as requested by the corresponding Ethical Approval).

Differences and novelty All automated SATD studies have been conducted on OO software and languages, many reusing the same dataset (Maldonado and Shihab 2015). Our study is distinctive because we explore SATD in scientific software, specifically R packages. Research regarding the identification or automation of SATD in scientific software, especially in R programming, is scarce. This contributes to one of the main novelties of this paper. As a result, we are contextualizing our work regarding other works in different domains.

R programming is innately different in terms of paradigm and construction (being dynamically-typed, derived from S, and package-based) and used for varied purposes (Storer 2017). However, scientific software cannot be compared to open-source development regarding the money spent on a project (Ahalt et al. 2014), the contributors' technical background (German et al. 2013; Pinto et al. 2018), and their formation (e.g., in scientific software, juniors contribute the most, and there are scarce third-party contributors) (Milewicz et al. 2019). Therefore, this study focused on an underdeveloped domain of knowledge.

As seen in the related work, most SATD automation has been done repeatedly on the same Java projects studied in the seminal SATD paper, regardless of it being over five years old. Research regarding the identification or automation of SATD in

scientific software, especially in R programming, is scarce. This contributes to one of the main novelties of this paper. As a result, we are contextualizing our investigation regarding other works in different domains.

Besides, existing studies on SATD in R did not cover automated techniques for its analysis (Codabux et al. 2021; Vidoni 2021b). Current SATD studies focus on ML or neural networks separately, without performing inter-algorithm comparisons. We conducted experiments on the three most-used ML algorithms, CNN, and PTM. Using PTM for the detection of SATD or SATD types has never been explored previously but were reported to be excellent for natural language processing in other areas of software engineering (Robbes and Janes 2019; Zhang et al. 2020). One study used BERT as the encoder template to remove obsolete to-do comments (which are a limited piece of the SATD spectrum of comments) from OO open-source code (Gao et al. 2021). Therefore, our study presents a more extensive usage, as it also compares two different BERTs.

Existing studies automate SATD versus non-SATD comments detection or focus on particular SATD types (e.g., Requirements). Our key difference is that we studied the detection of SATD automatically and investigated the efficiency of algorithms for the automatic detection of 12 types of SATD in R. Additionally, using those results, we assessed the causes of SATD from previous works in the OO domain (Mensah et al. 2018) and built on that to expand the corpus of causes.

3 Methodology

This section describes our goal, research questions, and the methodology used for our study.

3.1 Goal and research questions

This study aimed to automate the process of identifying SATD types by comparing available algorithms. The main aim is to determine SATD's plausible causes in R to enable the future development of tools to assist R developers. Thus, the following Research Questions (RQs) were pursued.

RQ1: Which technique has the best performance to extract SATD in R packages automatically? SATD in R packages is different than in OO. In prior studies, new keywords were uncovered for SATD in R compared to OO. (Vidoni 2021b; Codabux et al. 2021). Moreover, surveys confirmed R developers use source code comments differently, with a focus on TO-DO lists, rather than explaining the behaviour (Pinto et al. 2018). Moreover, there are specific SATD types that do not appear in OO (e.g., *Algorithm Debt*), and the distribution of occurrences is different (Vidoni 2021b). We explored the performance of techniques previously used for SATD detection and PTMs for identifying whether a comment in an R package is SATD or not. This question was used as a *preliminary step* to set the groundwork needed to answer the other RQs. Although using some of these algorithms for SATD detection is not novel, doing so for R packages and scientific software is.

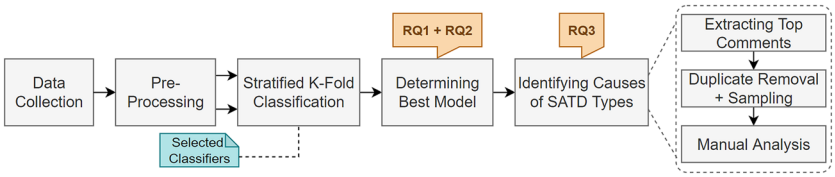

Fig. 1 Process of SATD detection and identifying causes of SATD types

RQ2: Which technique has the best performance in identifying different SATD types? Previous studies (Vidoni 2021b; Codabux et al. 2021) classified SATD in R packages as 12 different types, albeit manually or through semi-automated approaches. We were interested in determining which technique in RQ1 can classify the comments into different SATD types with the best performance. The goal of this question is to lay foundational grounds for future works.

RQ3: What are the causes leading to the occurrences of SATD in R packages? We explored why SATD occurs and what causes can be extracted from the comments. These are analyzed through hybrid card-sorting, starting with the categories defined by prior works (Mensah et al. 2018). We included all SATD types, unlike prior research, which focused mainly on a few (e.g., Code, Test, Design, Defect) (Ren et al. 2019; Zampetti et al. 2020; Wattanakriengkrai et al. 2018; Huang et al. 2018; Liu et al. 2018; Mensah et al. 2016).

3.2 Data preparation

This section describes the dataset, pre-processing steps, and methodology for classifying and identifying SATD causes. This process is depicted in Fig. 1, indicating which steps contributed to each RQ.

3.2.1 Dataset collection

The anonymized datasets we used were extracted and classified by a previous study (Vidoni 2021b), who mined 503 repositories of R packages publicly available on GitHub to extract source code comments (excluding documentation comments). Therefore, the datasets only contain source code comments. However, given that this was a mixed-methods study, the names of the packages mined were not distributed to protect the anonymity of the survey participants; this decision was enforced by the corresponding Ethical Approval, which translates to this current work. However, given the rigorous mining and labelling process, this dataset does not threaten the validity of this study. Since R does not have multi-line comments, Vidoni (2021b) used an R script was used to merge comments of several subsequent lines. Dataset D_1 is the result of a semi-automated classification of 164,261 comments. These comments are labelled and grouped into: *SATD* (about 4,962 instances) and *non-SATD* (159,299).

Commented-out code is source code which simply has a comment marker at the front, hence making it non-compilable/non-interpretable, and is not natural

language. Because of these intrinsic characteristics, one of the seminal works regarding SATD considered that commented-out code should be ignored as it "generally does not contain self-admitted technical debt" (da Silva Maldonado et al. 2017). This claim was also supported by prior studies (Potdar and Shihab 2014). Recent works have concluded that some commented-out code may be linked to 'on-hold' SATD (namely, implementations halted due to conditions outside of their scope of work) (Maipradit et al. 2020a), which is a different area of study and outside of the scope of this work.

As a result, since a percentage of our non-SATD sample were 'commented-out' code, we removed these comments, keeping 141,621 non-SATD comments used to answer RQ1 alongside the SATD comments. When removing the commented-out code, we also excluded the natural-language comments that appeared *inside the block of commented-out code*; the rationale for doing this was that these comments refer to no-longer-active code and would therefore provide incorrect statistics for our analysis. However, the threat of missing important information about the code's current state was negligible.

Though the dataset from Vidoni (2021b) does not name packages, their characteristics are known, given the description of their inclusion/exclusion criteria. D_1 is composed of packages published in/after, updated during/after 2018. These packages are public, open-source and have a minimal package structure (following CRAN's suggestions). A filtering step excluded personal repositories, deprecated/archived/non-maintained packages, data packages, and collections of teaching exercises or packages. The process was iterative, using control packages (e.g., `pkgdown`, `ggally`, `roxygen2`) to refine the selection.

D_2 is the *SATD* subset from D_1. It was classified and verified by Vidoni (2021b) using existing taxonomies into different types of TD; this was done by reading the comment for the related line of code. It typified the comments into *Algorithm, Architecture, Build, Code, Defect, Design, Documentation, People, Requirements, Test, Usability*, and *Versioning Debt*. Their definitions are summarised in Table 1; note that R had definitions adjusted by Codabux et al. (2021), hence why some types make specific clarifications.

However, the *D*2 dataset only included comments labelled as SATD, and we needed to determine the best technique for SATD classification among non-SATD (for RQ1). Therefore, we also wanted to detect non-SATD. To perform such an assessment, we randomly selected and added 2008 *non-SATD* samples from D_1 when conducting the experiments for RQ2, in order to have a sample of non-SATD to train the classifiers; this number was representative and calculated with 95% confidence and 5% error (using as population the whole of non-SATD comments, sans the commented-out code).

For RQ2, we added the non-SATD comments to evaluate the ability of the models for classification of types. However, as the recall and precision scores of the models are not 100%, meaning that there are cases of False Negative and False Positive (See Sect. 3.4 for definitions), we decided to add non-SATD comments in RQ2. Therefore, the models are evaluated when classifying different types of SATD and the non-SATD comments.

Table 1 Taxonomy TD definitions, based on (Codabux et al. 2021)

Debt type	Definition
Architecture	Refers to the problems encountered in product architecture, for example, violation of modularity, which can affect architectural requirements (e.g. performance, robustness)
Build	Refers to issues that make the build task harder and unnecessarily time-consuming. The build process can involve code that does not contribute to value to the customer. Moreover, if the build process needs to run ill-defined dependencies, the process becomes unnecessarily slow. When this occurs, one can identify Build Debt. In the context of R, Build TD encompasses anything related to Travis, Codcov.io, GitHub Actions, CI, AppVeyor, CRAN, CMD
Code	Refers to the problems found in the source code that can negatively affect the legibility of the code, making it more difficult to maintain. Usually, this TD can be identified by examining the source code for issues related to bad coding practices. In the context of R, code debt encompasses anything related to renaming classes and functions, ← and =, parameters and arguments in functions, `FALSE`/`TRUE` vs `F`/`T`, print vs warning/message
Defect	Refers to known defects, usually identified by testing activities or by the user and reported on bug tracking systems
Design	Refers to debt that can be discovered by analyzing the source code and identifying violations of the principles of good object-oriented design (e.g. very large or tightly coupled classes). In the context of R, design debt encompasses anything related to S3 classes and S4 methods, exporting functions with `@export` or the name pattern (visibility), internal functions with coupling issues, location of functions in the same file, selective importing `@import` (whole package) or `@importFrom` (a specific function), notations `::` and `:::`, returning objects (dataframes or tibbles), and Tidyverse vs. baseR
Documentation	Refers to the problems found in software project documentation and can be identified by looking for missing, inadequate, or incomplete documentation. In the context of R, documentation debt encompasses anything related to Roxygen2 (e.g., `@param` `@return`, `@example`), Pkgdown, Readme files, and Vignettes
Requirements	Refers to trade-offs made concerning 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 does not fully satisfy all the non-functional requirements (e.g. security, performance)
Test	Refers to issues found in testing activities that can affect the quality of those activities. Examples of this type of debt are planned tests that were not run, or known deficiencies in the test suite (e.g. low code coverage). In the context of R, test debt encompasses anything related to coverage, `covr`, unit testing (e.g., `testthat`), and test automation
Usability	Refers to inappropriate usability decisions that must be adjusted later. Examples of this debt are the lack of usability standards and inconsistency among navigational aspects of the software. In the context of R, this encompasses anything related to usability, interfaces, visualization, and so on
Versioning	Refers to problems in source code versioning, such as unnecessary code forks

The number of non-SATD comments was selected to approximately match the number of comments in the SATD type with the highest number of comments in the dataset (namely, the most 'common' TD type). Therefore, the resulting dataset can reflect the ability of the models in a more realistic setting, as SATD and non-SATD

Table 2 Statistics of dataset D_2 (Vidoni 2021b)

Abbr.	TD Type	# of Comments	% Total
CD	Code	2015	40.6
UT	Test	784	15.8
DF	Defect	693	13.97
REQ	Requirements	355	7.15
AR	Architecture	291	5.86
AL	Algorithm	276	5.56
DS	Design	221	4.45
BU	Build	160	3.22
US	Usability	71	1.4
DOC	Documentation	53	1.07
PP	People	21	0.42
VC	Versioning	21	0.42

are often blended in a dataset (i.e., a source code file can contain multiple and diverse instances of each). We did not consider a higher number of non-SATD comments in RQ2 as in RQ1, we study the models' capabilities to classify the SATD and non-SATD comments. As non-SATD comments cannot be detected by the models completely, and because the number of non-SATD comments is higher, we kept the number of non-SATD comments in RQ2 similar to the number of comments in the SATD type with the highest number of comments in the dataset to generate a balanced dataset. The statistics of the original dataset without the added *non-SATD* group are summarized in Table 2.

As mentioned, the dataset included comments from 503 R packages but the projects' information was removed by its authors to protect the package developers' privacy. Therefore, we cannot conduct analysis using within-project and cross-project settings like Wang et al. (2020).

3.2.2 Pre-processing

Following the previous studies on SATD (Flisar and Podgorelec 2019; da Silva Maldonado et al. 2017; Bavota and Russo 2016; Maldonado and Shihab 2015), we removed the punctuation from the comments, leaving exclamation and question marks (`!` and `?`, respectively) since these are shown to be helpful in SATD detection. All tokens were converted to lowercase, and we applied lemmatization using the Spacy library[2], as previously done in (Huang et al. 2018). Lemmatization was used to reduce the number of features when multiple formats of the same word (especially verbs) appeared in the comments, e.g., the word 'do' is used for all the variations 'done,' 'did,' and 'doing.'

[2] https://spacy.io/

Following the literature, we also removed stop words using the NLTK library, [3] but keeping the words mentioned in Huang et al. (2018); these are repetitive words such as 'as' and 'the' and are considered as noise features, especially when training ML models.

Eight comments from dataset D_2 and 181 comments from dataset D_1 were removed following these steps because they resulted in null (empty comment) after the stopword removals. Following the literature (da Silva Maldonado et al. 2017; Maldonado and Shihab 2015), we keep duplicate comments of each type, as they were associated with different code snippets. Two authors manually inspected the eight duplicate comments, alongside ten correctly cleaned, and assessed them regarding the original versions. This check was done to determine whether the removal was too aggressive. Both authors concluded in favour of the process–the comments had mostly filler symbols and were initially too short. There was no inter-rater agreement because the sample size was small (< 10).

3.2.3 Selected classifiers

We applied three main techniques to compare the results for both binary classification (D_1) and multi-class classification (D_2): i) traditional ML techniques, ii) deep neural networks, and iii) pre-trained neural language models (PTMs). In the first category, we used Max Entropy (ME), Support Vector Machine (SVM), and Logistic Regression (LR) classifiers. We applied a CNN and two PTM models (ALBERT and RoBERTa) as classification techniques for the second and third categories.

Max Entropy classifier is one of the first techniques in the literature for identifying SATD (da Silva Maldonado et al. 2017). This classifier is an ML model that enables multi-class classification and produces a probability distribution over different classes for each dataset item (Manning and Klein 2003). We chose SVM and Logistic Regression (LR) techniques since they are well-known classifiers for text classification and software engineering studies, including SATD detection (Kaur et al. 2017; Krishnaveni et al. 2020; Setyawan et al. 2018; Arya et al. 2019; da Silva Maldonado et al. 2017).

The number of studies applying neural network-based techniques for SATD detection is limited, and they often use either an RNN or LSTM architecture (Santos et al. 2020; Zampetti et al. 2018). Ren et al. (2019) applied CNN to identify SATD or non-SATD comments (Ren et al. 2019), while Wang et al. (2020) used attention-based Bi-LSTM. Among these, we chose the CNN approach for both datasets. The attention-based Bi-LSTM technique is not used here, as we ran two Transformer based models, which use attention mechanisms. This architecture has shown significant improvements over LSTM and RNN in many NLP areas, including classification tasks (Jiang et al. 2019; Naseem et al. 2020). The most recent SATD detection technique is from Wang et al. (2020), but it is not open-source, thereby hindering its use as a baseline algorithm.

[3] https://www.nltk.org/

In the third category, PTMs are language models trained on a substantial general-purpose corpus (e.g., book-corpus and wiki-documents) in an unsupervised manner to learn the context. Then, they are fine-tuned for downstream tasks (e.g., text classification, sentiment analysis) (Devlin et al. 2019; Liu et al. 2019). PTMs are known to reduce the effort and data required to build models from scratch for each of the downstream tasks separately (Liu et al. 2019) due to transferring the knowledge they have to other tasks. Therefore, we selected the following PTMs:

- ALBERT is a BERT self-supervised learning of language representation (Lan et al. 2020). The main reason for choosing ALBERT among other models is because it is a Transformer-based model. ALBERT is a lighter model, increasing the training speed of its base model BERT while lowering the memory consumption (Lan et al. 2020; Minaee et al. 2020). These are the main characteristics required to reduce the time required for training in the software engineering domain.
- RoBERTa stands for 'robustly optimized BERT approach,' which modified the pretraining steps of BERT (Devlin et al. 2019), outperforming all of its previous approaches to the classification tasks (Liu et al. 2019). RoBERTa is pre-trained on a dataset with longer sequences, making it a good candidate for our classification.

These models are based on the Transformer architecture, which uses attention mechanism (Devlin et al. 2019). This architecture is state of the art for many NLP tasks (Minaee et al. 2020) and has found its way into software engineering (Ahmad et al. 2020; Wang et al. 2019; You et al. 2019; Fan et al. 2018). ALBERT and RoBERTa have been previously used in the software engineering domain for sentiment classification (Zhang et al. 2020; Robbes and Janes 2019), and multiple text classification tasks for NLP (Minaee et al. 2020). However, they have not yet been used for SATD detection (or its classification into types), beyond the removal and detection of to-do comments (Gao et al. 2021). Therefore, we were interested in evaluating their ability to classify different types of SATD in the R packages, especially since we have a small number of labelled data for some TD types, and PTMs have proven exceptional for such cases (Liu et al. 2019).

3.3 Experimental setup

We experimented with balancing techniques using Synthetic Minority Oversampling Technique (SMOTE) (Chawla et al. 2002). Since the results of the ML classifiers for the balanced data using SMOTE decreased significantly compared to the imbalanced dataset, we did not use oversampling, but Weighted Cross-Entropy Loss (Phan and Yamamoto 2020; Wang et al. 2020). Cross-Entropy Loss is a general loss function in deep learning approaches to perform classification tasks since it shows better performance than other loss functions such as Mean Square Error (Zhang and Sabuncu 2018). However, for our imbalanced dataset, it might not have been optimal to use general Cross-Entropy Loss because the model's training would have

been inefficient, leading to the model learning useless information (Lin et al. 2020). Moreover, the loss function does not consider the frequency of different labels within the dataset since it treats loss equally for each label (Phan and Yamamoto 2020).

Therefore, to handle our case of imbalanced data, we applied Weighted Cross-Entropy Loss (Lin et al. 2020; Phan and Yamamoto 2020; Cui et al. 2019), which had been used previously for SATD detection (Wang et al. 2020). This method assigns weights to each label. Through this approach, the minority classes (those with less data) are given higher weights, and the majority classes (those with more data) get lower weights (Phan and Yamamoto 2020). Higher weights assigned to minority classes heavily penalize its misclassification, and the gradients are modified accordingly to accommodate minority classes. We used the class weights for traditional ML techniques to replicate this behavior in the other techniques we chose (e.g., SVM). The class weights parameters of the models are used to handle the data imbalance in the dataset. Class weights penalize the incorrect prediction made for a class/category A in proportion to the weight assigned to that class. Therefore, a high-class weight assigned to a category would penalize the mistakes more. The class weight for each class/category is set to the inverse of its frequency (i.e., their occurrence) in the dataset. Hence, the minority classes are assigned higher class weight values and would penalize the model more when an incorrect prediction is made for the minority class. This prevents the model from simply predicting the majority classes with high accuracy due to their higher number of samples in the dataset and prevents the model from overfitting.

For training the models, we applied weighted loss. First, we tune the hyperparameters of the models by splitting the data into 70% training, 10% validation, and 20% test. Then, we split the dataset into 80% train and 20% test sets to calculate the precision and recall values used to compare the performance of each algorithm (as needed for RQ1). Moreover, this also mitigated the chance of overfitting. To reduce the bias and variance related to the test set due to splitting both datasets D_1 and D_2, we used an alternate k-fold cross-validation technique called stratified k-fold cross-validation (Forman and Scholz 2010; Haibo He 2013).

K-fold cross-validation is a more rigorous approach for classification tasks, often used in software engineering (Zhang et al. 2020; Novielli et al. 2018). In k-fold cross-validation, dataset D_t is divided into k folds D_t^1, ..., D_t^k, with equally-sized divisions. K classifiers $c^{\mathbf{i}}$ are trained each time using one fold as test set, and the others are used as training set. Therefore, each split $D_t^{\mathbf{i}}$ is used as a test set once. Therefore, we will have k different performances on test sets. The stratified k-fold cross-validation is a common technique that reduces the experimental variance and creates an easier baseline to identify the best method when different models are compared together (Forman and Scholz 2010). This is similar to k-fold cross-validation, but the examples distribution for each class is maintained in each $D_t^{\mathbf{i}}$ split.

In particular, we considered $k = 5$, since this value has test error estimates without high bias or high variance, as disclosed by James et al. (2013). We also evaluated the methods with $k = 10$ since it has been used on prior works dealing with machine learning approaches to detect TD indicators (Siavvas et al. 2020; Cruz et al. 2020; Cunha et al. 2020). Therefore, for each dataset, we trained five classifiers for

each classification method and each DNN and each pre-trained model and reported their average evaluation metrics as explained in Sect. 3.4. All experiments ran on a Linux machine with Intel 2.21 GHz CPU and 16G memory and used each model's publicly available source code.

We tested multiple features to train the SVM and LR, including Term-Frequency Inverse Document Frequency (TF-IDF), Count Vectorizer, and neural word embeddings. However, the Count Vectorizer performed better than TF-IDF in both cases. We also experimented with unigram, bigram, and trigram independently to choose features, but the unigram-bigram combination gave the best results as reported in this study.

Moreover, we experimented with multiple settings to train the CNN since the literature was scarce for SATD. The best results are reported here. We trained the CNN with 30, 50, and 100 epochs and used 0.3, 0.5, and 0.7 dropout values and 32, 64, and 128 batch sizes. The final setting was 50 epochs, 0.5 dropout value, and 64 for the batch size. We experimented with ALBERT-based and RoBERTa-based models, each with 12 encoder layers. To fine-tune it to our dataset, we used the HuggingFace library[4] along with training and validation losses to ensure the models were not overfitting nor underfitting (the latter for the deep learning models). The training data used for training ML techniques was also used in the fine-tuning.

Finally, following previous works (da Silva Maldonado et al. 2017; Bavota and Russo 2016), we conducted the manual analysis on the documents that are identified to include the words most contributing to a SATD type (instead of the whole dataset). In machine learning algorithms, the contributing words to each SATD class, and thus the documents containing them, can be extracted using the features' weights in the algorithm. However, finding the words that have the highest contribution for each of the SATD type classes is not straightforward in deep learning models. To accomplish this, we extracted the attention values from the RoBERTa model, used to score the comments in each class for RQ3. HuggingFace provides a sequence classification head on top of the pooled output, which is the hidden state of the first token of the sequence, processed by a classification head that is a linear neural layer. To calculate the importance of the tokens, we extracted the attention scores for each unique token in the train set, choosing the last layer from the stack of encoders because it is the same layer used by the classification head. Inside the encoder, each layer consists of 12 attention-heads. Therefore, to incorporate information from all the heads, we added the attention value of a unique token from each attention-head, assigning it to the unique token. We repeated this process for all the training samples and obtained the values for each token.

The tokens with the highest attention score during the fine-tuning of the model helped understand which tokens encoded important information to classify each type (namely, identifying meaningful keywords per type). Therefore, these values were used to rank the comments for each class of interest and are further discussed in Sect. 3.5. Attention values were collected independently for each SATD type using their respective datasets.

[4] https://huggingface.co/

3.4 Evaluation metrics

The following metrics were used in the classification.

Precision (P) Precision divides the number of records predicted correctly to belong to a class (TP) by the total number of observations that are predicted in that class by the classifier (TP + FP): $P = \frac{TP}{TP+FP}$. Here, TP means True Positive, and FP means False Positive (i.e., the number of observations incorrectly predicted to belong to a class). In multi-class classification, the FP for each class C is the total number of records with other labels that the classifier predicted to belong to class C.

Recall (R) Recall is calculated by dividing the number of observations correctly predicted to belong to class C (TP) by the total number of records in the corresponding class: $R = \frac{TP}{TP+FN}$. Like before, FN stands for False Negative, representing the number of records in a class that the classifier incorrectly predicted to belong to other classes.

F1-Score (F1) The F1-Score is computed as $F1 = \frac{2\cdot(P\cdot R)}{P+R}$. It shows the weighted average of Precision and Recall. As we used 5-fold cross-validation for RQ1, we reported the average scores for each of the P, R, and F1 scores over the $k = 5$ classifiers:

$$F1^{\text{avg}} = 1/k \sum_{n=1}^{k} F1^{(\mathbf{i})} \tag{1}$$

where for D_t, $F1^{(i)}$ is the performance of classifier $c^{(i)}$ on test set $D_t{}^{(i)}$. Likewise, the average of P^{avg} and R^{avg} were reported.

For multi-class classifiers (RQ2), especially with imbalanced data, we used micro-average and macro-average metrics of P, R and $F1$ scores. The micro-average weights the contribution of the class with the predominant number of records. The macro-average takes the average of the scores for each class and weights them equally in the final score. The micro- and macro-average Precision are calculated as follows:

$$P_{micro} = \frac{\sum_{j=1}^{m} TP_j}{\sum_{j=1}^{m} TP_j + \sum_{j=1}^{m} FP_j} \tag{2}$$

$$P_{macro} = \frac{\sum_{j=1}^{m} P_j}{m} \tag{3}$$

TP_j and FP_j are the number of TP and FP for the j-th class. P_j is the precision computed for class j and m is the number of classes. Similar calculations led to the micro average and macro average of Recall and F1-score, denoted as R_{micro}, R_{macro}, $F1_{micro}$, and $F1_{macro}$. For the stratified 5-fold cross-validation, we report the averages of these metrics over the $k = 5$ classifiers, calculated as follows:

$$F1^{avg}{}_{micro} = 1/k \sum_{n=1}^{k} F1^{micro}{}_{(i)} \tag{4}$$

$$F1^{avg}{}_{macro} = 1/k \sum_{n=1}^{k} F1\_macro^{(i)} \tag{5}$$

For dataset D_t, the $F1_{micro}{}^{(i)}$ and $F1_{macro}{}^{(i)}$ are the performance of classifier $c^{(i)}$ on test set $D_t{}^{(i)}$. Similar calculations were done to compute $P^{avg}{}_{micro}$, $P^{avg}{}_{macro}$, $R^{avg}{}_{micro}$, and $R^{avg}{}_{macro}$.

Following Zhang et al. (2020), we considered a model to have better performance on D_t if it had higher values in both $F1^{avg}{}_{micro}$ and $F1^{avg}{}_{macro}$ scores.

3.5 Manual analysis

This section discusses the required sample size calculation and the classifications used.

3.5.1 Sample size calculation

We used the results of RQ1 and RQ2 to determine the technique that outperforms other models when classifying comments in R packages. Based on the results, we manually identified the causes for SATD introduction (for RQ3) from dataset D_2, including different SATD types. For the ML techniques, we followed prior works (Flisar and Podgorelec 2019; Mensah et al. 2018). The most contributing features (keywords) for each type were extracted using the model's results, i.e., those with the highest weights in each class of interest (i.e., SATD type). Then, we extracted the comments that had those features/keywords. For dataset D_2, the identification was completed for each class separately. For the neural network models, the technique was slightly different–as CNN results are lower than PTMs', we only explained the process for PTMs. The Transformer models use an attention mechanism to determine which tokens in the comments should be given relatively higher importance (Devlin et al. 2019).

To extract the critical comments in each category, we used the attention values of the words–namely, the values generated by the models. This value was used to find the top words the model attended for each SATD type (i.e., extracting the words with the highest attention score). For each comment, we summed up the attention value of its tokens to calculate the total attention value of the sentence. However, longer sentences could have higher values with this approach since they had more tokens to sum (including meaningless words). Therefore, we normalized this score by dividing it by the total number of tokens present within the sentence (effectively turning it into a proportion per word). We used the mean attention of sentences and sorted them based on their average. The ranked comments show those with the highest attention by the model for each SATD type in order, similar to extracting features with the highest weights in ML techniques. The details of the process used for extracting attention values for each token are discussed in Sect. 3.3.

As the number of comments is significant, we identified a representative sample from the ranked comments for manual analysis. Since the ranked comments were duplicated for some types (e.g., `nocov` comments in *Test Debt*), we used distinct

Table 3 Dataset D_2 after removing duplicates and sampling (see Table 2) for statistics)

Abbr.	TD Type	# w/o Duplicates	Sample Size
CD	Code	1232	293
UT	Test	281	163
DF	Defect	474	212
REQ	Requirements	220	140
AR	Architecture	195	130
AL	Algorithm	180	123
DS	Design	142	104
BU	Build	97	78
US	Usability	45	40
DOC	Documentation	39	35
PP	People	16	15
VC	Versioning	12	12
	Total	2933	1345

comments only for the manual analysis and obtained the sample size after removing duplicates. We searched for a representative sample of size n per type and calculated it with a confidence level of 95% and confidence interval of 5 for *each* SATD type (namely, we obtained 12 samples, one per type). [5] Although the sample was not randomly selected, by using this process, we ensured that it represented the whole set. As we have the ranked list of comments (computed from the highest attention scores from the DL models), we used the n comments that have the highest scores in each SATD type for the manual analysis. Overall, the 12 samples accounted for 1345 Comments in total for the manual analysis; the statistics of D_2 and the per-type sample size are available in Table 3. Even if the reduction in some types was minor or non-existent (e.g., *Usability Debt* or *Versioning Debt*), we worked with the sample to have a standardized approach across all SATD types.

Finally, random sampling was not required because the comments are ranked, and the threat of missing some essential ones by only using the high-ranking comments was negligible.

3.5.2 Identifying causes

Two of the authors read the comments for each SATD type to identify the plausible causes behind each SATD comment, using the categories identified by Mensah et al. (2018). These are: The causes they identified were: *Code Smells* (violated methods and classes resulting in serious problems in a project, according to Fowler's Code Smell definition[6], *Time Constraints* (actions are limited by deadlines), *Too Complicated and Complex* (developers resorting to simple solutions because of the

[5] The sample size was calculated using https://www.surveysystem.com/sscalc.htm.
[6] Available on his site: http://martinfowler.com/bliki/CodeSmell.html).

complexity, effort or knowledge needed otherwise), *Inconsistent Performance* (the software's performance varies for the same function), *Inconsistent Communication* (clear misunderstandings, hearsays, confusing communication), *Heedless Failure to Remember* (developers cannot remember what they were supposed to do, or they are writing them down not to forget), *Inadequate Testing* (a variety of issues related to unit testing).

However, Mensah et al. (2018) studied only *Code Debt*, without considering other SATD types. Therefore, we extended their corpus to include new causes not uncovered before.

To classify the 1,345 comments, we used a *hybrid card-sorting technique* (Whitworth et al. 2006), which is a combination of open and closed card sorting. We started the classification using the SATD types from Mensah et al. (2018) (*closed card-sorting*), but detected 'emergent' cause as we went along. Therefore, we conducted an *open card-sorting*; when we detected an emergent cause, we gathered several comments possibly fitting that cause, discussed them, and decided whether a new cause was warranted or the comments fit existing causes. In cases of new causes, we decided on a name and acronym and the conditions to label it. Then, we repeated the individual labelling using these new categories and peer-reviewed the final results to determine the agreements.

Finally, we obtained 16 plausible SATD causes summarised in Table 4. Note that the categories explained in this Table also include results from this paper (present in the fourth 'block' of the Table). We chose to present this partial information here for ease of reading, but further discussions will be presented in Sect. 4.3.

Two authors separately classified the comments according to the resulting 16 SATD plausible causes. Note that some comments pertained clearly to more than one cause and were thus categorized using multiple causes. Both authors discussed disagreements through peer-review sessions to finalize the causes of each SATD instance. The inter-raters' reliability was calculated using Cohen's Kappa coefficient (McHugh 2012); this is a test that measures the level of agreement among raters and is a number between -1 (highest disagreement) and $+1$ (highest agreement). On average, the manual classification of this step led to an agreement of 83.04%, which is considered high. The resulting manual SATD classification dataset is publicly available for replication purposes.[7]

4 Results

This section presents our results to the RQs.

[7] https://bit.ly/3sGqkRe

Table 4 Existent and new causes for SATD used and identified in this manuscript

Cause name	Acronym	Notes
Failure to Remember	HFR	These causes were named and defined by Mensah et al. (2018), and reused as-is
Complicated or Complex	TCC	
Inconsistent Communication	IC	
Unexpected Performance	UCP	
Time Constraints	TC	
Insufficient Testing	INT	Originally named 'Inadequate Code Testing' (ICT) (Mensah et al. 2018). It was divided into INT and TNW to specify it
Tests Not Working	TNW	
Workarounds or Hacks	WOH	The category 'Code Smells' of (Mensah et al. 2018) was renamed for clearer understanding
Dead or Unused Code	DC	These are the new causes uncovered in this work, through the hybrid card-sorting process disclosed in Sect. 3.5
Missing or Incomplete Features	MIC	
Misunderstanding Features	MOF	
Known Bugs	BUG	
Warnings to Developers	WTD	
Developers' Questions	DQ	
Instructions and Steps	IAS	
Lack of Knowledge	LOK	

Table 5 Classification results on dataset D_1 (RQ1)

Approach	SATD (%)			
	P^{avg}	R^{avg}	$F1^{avg}$	Training time
ME	78.88	74.02	76.36	1 min 52s
SVM	64.62	70.05	67.22	1 min 20s
LR	61.04	72.74	66.37	0 min 16s
CNN	83.92	76.29	79.89	3 min 18s
ALBERT	**87.62**	85.03	**86.21**	52 min 6s
RoBERTa	85.91	**86.27**	86.09	48 min 58s

The best results are given in bold

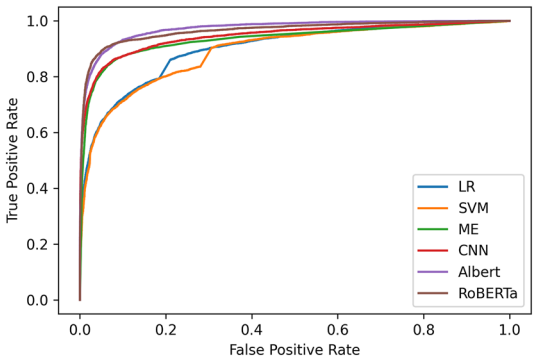

Fig. 2 AUC plot of models for SATD detection (binary classification)

4.1 RQ1: Techniques to extract SATD

The results of the classifiers applied to dataset D_1 are presented in Table 5. We report the average of five classifiers in the 5-fold stratified cross-validation for Precision (P), Recall (R), and $F1$ scores for each model. The training time for the models is presented in the last column of Table 5. Although there is a significant difference among the models' training time, the inference times are close to each other, 1/268 of a second for CNN and in the range of 1/96 to 1/90 of a second for all other models.

Among the three categories of the models we studied, PTMs perform the best.

Overall, in the ML group, Max Entropy had the best results for all scores, compared to SVM and LR. However, the results of SVM were slightly better than LR. ME outperforms SVM by 14.28, 3.95, and 9.18 scores in P^{avg}, R^{avg}, and $F1^{avg}$. PTM improved the ME results by about 9.73 F1 score (namely, they performed significantly better). CNN performs better than Max Entropy, but its average precision, recall and $F1$ scores were 3.68, 9.98, and 6.31 lower than the best performing PTM (thus, it outperformed Max Entropy, but not PTMs). ALBERT performed slightly better than RoBERTa in two scores, and improved ME results by $P^{avg} = 11\%$, $R^{avg} = 14.8\%$, and $F1^{avg} = 12.8\%$ scores, respectively.

These numbers were compared to the work of Zampetti et al. (2020), who reported reaching up to 73% precision and 63% recall for removing SATD using their patterns combining Recurrent Neural Networks and CNNs, and the

work of Ren et al. (2019) which achieves average F1-score in the range of high 70s in their experiments using CNN for the detection of SATD and non-SATD. Though the datasets are different in these works, the CNN performs around 80% F1 score in our case, which is still lower than the performances of ALBERT and RoBERTa.

If we consider the non-SATD category as the main class to be predicted, the F1 score for all models were above 90. For the others, the F1 scores were $ME = 95.14$, $SVM = 91.59$, $LR = 92.34$, and $CNN = 95.90$, while for the PTMs, the F1 scores were $ALBERT = 97.09$ and $RoBERTa = 96.83$, respectively. In particular, ME, CNN, and the pre-trained models are in the high 90s.

As RQ1 discusses the binary classification, we also provide the Area under the ROC Curve (AUC) plots of the models as presented in Fig. 2. AUC value is a number between 0 and 1, inclusive, and a higher number shows the model has a good measure in separating the SATD class. The values of the AUC plots are, in ascending order, $SVM = 0.89$, $LR = 0.90$, $ME = 0.93$, $CNN = 0.94$, $ALBERT = 0.97$, and $RoBERTa = 0.97$. These results confirm that both PTMs perform better than other models and that the results of CNN and ME are close but slightly behind the PTMs' AUC values. As a result, CNN and ME could be a less expensive implementation model (in terms of computational resources).

PTM models better predict SATD comments, having higher Recall, Precision and AUC values. Note that this question was a binary classification into SATD and non-SATD to prepare the datasets for RQ1. As a result, this first step did not distinguish between SATD types.

RQ1 Findings. Max Entropy (ME) classifies SATD comments in R packages with an $F1 > 76\%$, and CNN improves the ME results slightly. Besides, PTMs increase ME's results up to 10% F1 score and achieve the best performance.

4.2 RQ2: Techniques to detect SATD types

The results of our experiments on identifying different types of SATD is presented in Table 6. The highest score among all models is shown in bold. The highest scores among the models within the ML category are underlined. ALBERT displays results for 10 and 30 epochs. We only report the average F1 scores for all the models.

This analysis produced similar results to RQ1. Among the ML models, ME performed better than SVM and LR models, while PTM outperformed the other models. Following the literature (Liu et al. 2019; Kanade et al. 2020), we fine-tuned ALBERT and RoBERTa models for ten epochs, since training RoBERTa for more epochs leads to overfitting.

Based on the results obtained for ALBERT for *People* type (which is 0), we decided to retrain ALBERT for 30 epochs (present as ALBERT-30 in Table 6). This improved the results for all types, especially for those with a small number of comments in the test set; e.g., for *People*, the F1 score progressed from 0 to 52.82, which is remarkable given the low number of cases present in the dataset. The exception for this improvement was two categories (namely, *Requirements* and *Algorithm*

Table 6 Classification results on dataset D_2 (RQ2)

SATD Type	$F1^{avg}$ (%)						
	ME	SVM	LR	CNN	ALBERT-10	ALBERT-30	RoBERTa
Testing	83.24	82.42	84.07	84.68	87.42	**87.81**	86.88
Code	65.96	54.91	53.15	63.42	67.53	67.99	**68.56**
Versioning	44.76	46.43	51.75	48.00	38.23	41.42	**61.43**
Architecture	47.50	39.51	41.77	50.04	53.61	57.80	**58.14**
Defect	49.28	46.70	49.30	49.76	56.34	**58.27**	57.66
Build	48.22	46.39	41.94	46.47	38.69	43.35	**52.06**
Documentation	49.76	32.32	39.15	0	21.05	45.97	**51.26**
Requirements	37.86	38.17	39.82	35.92	42.40	*40.27*	**46.62**
Design	44.46	34.47	37.74	33.27	30.87	31.69	**45.37**
Usability	38.56	35.30	32.56	23.77	36.58	37.63	**43.06**
People	34.68	7.34	10.6	0	0	52.82	**42.29**
Algorithm	28.48	23.58	25.27	23.09	24.78	*24.02*	**31.30**
Non-SATD	79.18	75.64	76.27	82.18	88.26	**88.12**	87.76
Micro-avg	65.64	59.19	58.93	64.21	68.58	69.40	**70.94**
Macro-avg	50.15	43.32	44.77	41.58	45.04	52.09	**56.34**

Debt) that had a slight decrease in F1 score; we believe this was because of how varied comments of these two types were, but a more detailed analysis was out of the scope of this study.

Although the micro-average did not improve much between ALBERT-10 and ALBERT-30, the macro-average increased by 7 scores in the latter. Among all the models, RoBERTa has the best performance for all types, which is different from the results of RQ1. For detecting SATD versus non-SATD comments, ALBERT-10 had slightly better Precision and F1 scores. We did not perform another assessment with more than 30 epochs due to the risk of overfitting.

However, to distinguish between the different SATD types, RoBERTa is a better model as it achieved a better performance in most SATD types (except *Test* and *Defect Debt*), with only ten epochs. For a binary SATD/non-SATD classification, ALBERT-30 provided the best outcome.

Similar to RQ1, SVM and LR had the lowest scores. Max Entropy performed better than CNN since it had higher micro- and macro-average F1 scores than CNN; this can be related to the large amount of data that DL models require for training. Interestingly, CNN results for *Documentation* and *People Debt* are 0; this may be due to a combination of sample size and language variability (meaning, how many diverse words are used in the comments), but a detailed analysis remained out of scope. Nevertheless, Max Entropy detected comments of these two types with 49.8 and 34.7 F1 scores. RoBERTa improved the results of Max Entropy by 5.34 micro-average and 6.24 macro-average, which was an 8% and 12.3% improvement of the results, respectively.

It is worth mentioning that ME was previously used for identifying *Design* and *Requirements Debt*, where the authors report an average F1-score of 62% and 40.3% for each category in a Java dataset, respectively (da Silva Maldonado et al. 2017). Although the datasets are different between their and our study, ME achieves average F1 scores of 44.46% for *Design Debt* and 37.86% for *Requirements Debt* in our dataset, which are lower than the previously reported numbers. Even the best performing models in our study have F1 scores of 45.37% and 46.62% for *Design Debt* and *Requirements Debt* in R, respectively. The number is still below the reported number for *Design Debt* reported in da Silva Maldonado et al. (2017), which can be an indication of the differences in SATD occurrences in OO programming and R.

Prediction difficulty. Interestingly, there is a difference in the ability of the models to predict the type of SATD comments. Based on the results, some types have higher, and others have lower F1-scores. For example, considering RoBERTa's results, the performance is higher for non-SATD, followed by *Test Debt* with a significant margin compared to all other types; however, this type was fairly 'standardized', meaning that the comments revolved around limited and repetitive issues. After that, the performance in descending order belongs to *Code*, *Versioning*, *Architecture*, *Defect*, *Build*, *Documentation*, *Requirements*, *Design*, *Usability*, *People*, and *Algorithm Debts*. Note that the last five SATD types have F1 scores below 50%. Among these, the lower the scores, the more distinct keywords were used in those comments.

Although a detailed investigation regarding the nuances of the natural language was out of scope for this paper, we did notice several characteristics that may be influenced the F1 scores. There are three cases worth discussing:

- *Low sampling numbers:* Two types, namely *Usability* and *People* Debt have 71 and 21 samples respectively, as seen on Table 2. As a result, the low F1-score of these cases can be assumed to be caused by the low training sample. Note that these TD types are not as common as others and have not been studied in other SATD investigations of automated detection (Vidoni 2021b).
- *Adequate sample, variable vocabulary:* This is the case of *Requirements*, *Design* and *Algorithm Debts*, in which the original samples are 355, 221 and 276 respectively, but the F1-scores all remain below 50. However, given the nature of these TD types, the vocabulary present in the sample comments is nuanced and highly variable–this means that, although it is easy for a human to detect them, the algorithms are not. For example: **`Need to create a separate image for every different vertex color`** is Requirements Debt. However, **`currently only for a single layer`** and **`nothing for seasonality yet`** are also Requirements Debt. They do not always have keywords (e.g., there is no 'to-do'), and the language can be ambiguous (e.g., the word 'layer' was referring to layers of a plot in `ggplot2` style, and not to architectural layers). Note that investigating the semantics of the natural language used for each TD type in scientific software was out of scope for this study; this would also need dividing scientific software into domains (e.g., bioinformatics, geology, general statistics) to further narrow down the language.

Table 7 Classification results of max entropy on D_2, with and without lemmatization (Lemm). best result is underlined

SATD Type	$F1^{avg}$ Score (%)	
	ME (with Lemm.)	ME (without Lemm.)
Code	65.96	66.96
Test	83.24	85.28
Defect	49.28	53.18
Requirements	37.86	39.82
Architecture	47.50	47.60
Algorithm	28.48	29.78
Design	44.46	41.62
Build	48.22	46.96
Usability	38.56	43.34
Documentation	49.76	52.50
People	34.68	29.34
Versioning	44.76	44.76
Non-SATD	79.18	83.96
Micro-average	65.64	67.95
Macro-average	50.15	51.16

- *Low samples, relatively stable vocabulary:* In this case, *Versioning* and *Documentation* also have very low samples (21 and 53, respectively), but the F1-scores are higher. However, they both represent different situations. Regarding *Versioning Debt*, the samples have a stable jargon, and it is also highly possible that the models were overfitted; we did not detect this when revising the selection, and it is not easily done either as samples are scarce, it does not seem to be reported in source comments (as per the survey by Vidoni (2021b); Pinto et al. (2018)), and it has not been automatically detected in source code comments before. Regarding *Documentation Debt* the vocabulary is somewhat more stable, but given that F1 is about 51%, we could consider the lower performance as a straightforward case of low samples.

When comparing these numbers to the statistics of Table 2, there is no relation between the number of available comments in the dataset and the performance of the models in predicting their types. For instance, the F1 score for *Versioning* is 61.43%. However, comments labelled as *Requirements* contain 7.15% of the comments in D_2, but the F1 score for this type is 46.62%. Likewise, *Test Debt* has higher scores than *Code Debt* in all the models, although the number of comments for *Code* are 2.5 times the number of comments labelled as *Test Debt*. As a result, we can conclude that the models' performance was not linked to the number of available comments for a particular SATD type.

Table 8 Frequency (as a proportion of occurrences) for the causes to introduce SATD in R packages. some comments have more than one reason

Cause	Frequency by Type of Debt											
	AL	AR	BU	CD	DF	DS	DOC	PP	REQ	UT	US	VC
HFR	**47.97**	*40.00*	*24.36*	**44.37**	*20.75*	**44.23**	**60.00**	**46.67**	**71.43**	**34.36**	**50.00**	**41.67**
TCC	5.69	3.08	1.28	7.17	1.89	16.35	0.00	6.67	0.00	0.61	5.00	0.00
WOH	2.44	7.69	**48.72**	*15.70*	4.25	11.54	2.86	0.00	1.43	21.47	0.00	0.00
DC	0.00	0.77	0.00	8.53	0.47	0.00	0.00	0.00	0.00	1.87	0.00	0.00
MIC	15.45	1.54	1.28	5.46	7.08	4.81	2.86	0.00	*32.14*	3.07	*27.50*	16.67
MOF	4.88	0.00	0.00	0.68	1.89	2.88	0.00	0.00	5.00	1.23	5.00	8.33
INT	0.00	0.77	0.00	2.05	0.00	0.00	2.86	0.00	0.00	*22.70*	0.00	0.00
TNW	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	7.98	0.00	0.00
BUG	1.63	4.62	5.13	2.73	**45.28**	3.85	0.00	6.67	3.57	4.29	5.00	0.00
WTD	10.57	16.92	12.82	4.44	9.43	8.65	5.71	0.00	0.71	5.52	2.50	0.00
DQ	*26.83*	3.85	2.56	10.92	10.38	8.65	8.57	*20.00*	13.57	4.91	12.50	8.33
IAS	4.07	1.54	3.85	1.37	4.25	*19.23*	*20.00*	0.00	3.57	4.29	7.50	*33.33*
IC	1.63	0.77	0.00	0.34	0.00	0.96	5.71	**46.67**	0.00	0.61	0.00	0.00
LOK	3.25	4.62	2.56	7.85	4.72	2.88	0.00	0.00	2.86	4.91	2.50	0.00
UCP	1.63	**44.62**	2.56	0.34	1.42	0.00	0.00	13.33	0.00	0.61	0.00	0.00
TC	5.69	3.08	1.28	7.17	1.89	16.35	0.00	6.67	0.00	0.61	5.00	0.00

4.2.1 Effect of lemmatization

ML approaches require feature engineering and reduction. One of the main techniques applied for reducing the number of features is lemmatization. As there are no previous studies on the effect of lemmatization on SATD detection, we conducted another study. We chose the best performing ML model (i.e., Max Entropy) and trained it on the dataset without lemmatization. The goal was to assess if the results changed compared to those reported in Table 6 and if they could close the gap with PTMs' performance.

For dataset D_1, when ME is used without lemmatization, the scores are $P^{avg} = 81.9$, $R^{avg} = 78.2$, and $F1^{avg} = 80.0$. Moreover, the results on dataset D_2 without lemmatization improved by increasing the F1 score, as reported in Table 7. The exception to this are *Design*, *Build* and *People Debt*. A possible reason for this could be due to the variety of words used by developers, as some of them belong to different SATD types.

Comparing these results to PTM models, ME's outcomes were still surpassed by 4.4% and 10% on micro- and macro-average scores by the RoBERTa model, even with lemmatization. Moreover, RoBERTa has the advantage of higher scores for SATD types with few labelled comments, such as *People* and *Versioning*. However, PTM models take longer to train and require more computational power than ML models; thus, a lemmatized ME model could be preferred if time and computational power are an issue.

RQ2 findings Similar to RQ1, the best models to classify different SATD types in R packages remained PTMs and ME. The best performing model was RoBERTa, which worked better in types with few comments for training. We relate this to the knowledge learned by PTMs during their pretraining on the large general-purpose text.

The major findings for this question were:

1. Prior works had only successfully detected 5 SATD types, and we detected 12 SATD types without overfitting the models.
2. We used PTMs for the first time for SATD *per-type* detection and demonstrated that they outperformed other techniques even in cases with few available samples.
3. Regarding ML models, our study to determine the effect of lemmatization in Max Entropy is the first of its kind. It demonstrated positive results in most SATD types, provided they have a large sample. However, though the $F1$ values improve, they do not match the PTMs performance.

4.3 RQ3: Causes of SATD types in R

The comment classification phase categorized the 12 SATD types according to 16 plausible causes as shown in Table 8. Comparing the types to the causes of introduction is necessary to explore the nuances between TD types–i.e., before the classification, it was possible to hypothesize that some TD types may have different causes (e.g., reasons) for happening/being introduced.

The process to obtain these causes of TD introduction was discussed in Sect.3.5, and the full names for each acronym were presented in Table 4. Note that some comments have multiple causes, thus, the proportions exceed 100 when summed up. For readability, Table 8 highlights extreme cases; the most common cause is highlighted in bold red, the second-most in *italics orange*, a special case (discussed below) in underlined green.

As mentioned in Sect. 3.5, seven of these categories were defined and uncovered by Mensah et al. (2018). However, the category 'Inadequate Code Testing' was divided into 'Insufficient Testing' (INT) and 'Tests Not Working' (TNW), to differentiate between different types of test smells, given that prior studies demonstrated that R packages are prone to *Test Debt* (Vidoni 2021a). Thus, we count INT and TNW as the 'original' eight categories. This distinction was possible since we had a large sample of *Test Debt* and a prior study that identified different types of test smells (Vidoni 2021a). Likewise, the original category of 'Code Smells' was renamed 'Workarounds or Hacks' (WOH) due to the original name being too generic. These changes were summarised in Table 4.

We worked with these causes as a starting point. However, given that OO (or traditional, commercial) software has some differences with scientific software, through the process described in Sect. 3.5, we identified eight *new* causes of SATD (also presented in Table 4). However, while we detected these in the source code comments of R packages, they are generic enough to be extended to

other languages. Further research is required to validate these causes in the context of other languages. Overall, the new types are:

- 'Dead or Unused Code' (DC), which supports the findings of Vidoni (2021b), who identified a concerning trend of R developers to leave unused code commented-out rather than removing it. Moreover, this is also a known smell for *Code Debt* (Kaur and Dhiman 2019), but it had not been identified as a cause for SATD before.
- Vidoni (2021b) surveyed developers about the comments they wrote in their R packages and determined that many of them add notes about bugs or issues as reminders to work on them but seldom address those comments. Those prior findings align with the new causes 'Known Bugs' (BUG), 'Warning to Developers' (WTD), and 'Developers' Questions' (DQ).
- 'Missing or Incomplete Features' (MIC), 'Misunderstanding Features' (MOF) and 'Instructions and Steps' (IAS) also support prior works that surveyed R developers and determined they use source code comments to document possible features, rather than perform thorough elicitation (Pinto et al. 2018).
- 'Lack of Knowledge' (LOK) is also considered a cause, as it aligns with prior studies that determined some *inadvertent* developers can incur in TD unintentionally (Fowler 2009; Codabux et al. 2017). This is also related to 'Developers' Questions' (DQ), especially in collaborative software development (namely, a comment is added hoping that another developer works on it).

Regarding frequencies, 'Failure to Remember' (HFR) is either the most common or second-most common cause for all SATD types, with only three types having it as the second-most common cause (i.e., *Architecture*, *Build* and *Defect*). This aligns with the survey results that accompanied the original paper that provided the dataset (Vidoni 2021b) since the participants disclosed that they mostly add SATD as self-reminders. Moreover, since we allowed a comment to be classified into more than one cause, several had HFR as a secondary reason. HFR is one of the most 'flexible' causes of SATD, meaning it applies to multiple cases and can be combined with other causes. The following are some post-processed examples of comments:

- *Architecture:* **`mean for any k dim array fixme ? reduce is very slow here`** related to a performance issue (UCP) but was a still-unresolved fix request (hence, HFR since it could have been forgotten and not fixed). It was not a BUG because the function was working, but with lower performance. Also, **`make sure no segment of length 1 remains to-do this should not occur and needs to be prevented upstream`** was classified as HFR since it was a clear reminder of a task left for the future.
- *Documentation:* **`profile fixme how to handle the noise if as below document it`**, is an HFR because it is a reminder to update the documentation as per the comments explaining the code below.

- *Test:* **`to-do add logging more in-out tests add test case in package`** indicated the need to add more tests (INT), but was also something not yet done, and flagged as a reminder for the future, hence HFT.

Note that we could not check whether some of the HFR had been successfully fixed in a future commit, and it was not feasible to elicit whether a developer forgot about a comment or not. Nevertheless, it was deemed that, while a comment (possibly categorized as HFR) was left in the code, it meant the developers had not yet fixed, thus being 'temporarily forgotten'. Therefore, though there is a chance the occurrence was slightly higher, prior works also identified HFR as the most common cause (Mensah et al. 2018), thus supporting our findings.

After that, 'Instructions and Steps' (IAS) was the second-most common cause for three debts (namely, *Design*, *Documentation* and *Versioning Debt*), which as explained above, matched behavioural findings of other studies (Pinto et al. 2018; Vidoni 2021b). Relevant examples are **`cell has to be a list column for the tibble add row in insert shims to work with increased type`** (*Design Debt*, IAS only as it explained how to use a particular parameter), and **`roxygen2 can t overwrite namespace unless it created it so trick it into thinking it did and add the rstan`** (*Documentation Debt*, IAS only as it was explaining to other developers why a piece of code was added). These comments often explained why the debt was introduced or how to work with it instead of fixing it.

Finally, TNW is a particular case to be discussed since it was only found, as expected, in comments containing *Test Debt*. It was used to indicate when a test was not working and remained unfixed. For example, **`to-do rework these tests because currently failing`** (HFR, TNW), and **`fails when plpresult is not class plpresult`** (TNW). Finding this cause for SATD introduction only on *Test Debt* was expected since it is inherent to this particular debt type.

RQ3 Findings We uncovered eight new causes of SATD that support prior findings on related areas and increased the number of plausible causes for SATD introduction from eight categories to 16. Though this study worked with R packages, the causes are generalizable enough to be studied in other programming languages.

Regarding frequencies, we confirmed that 'Failure to Remember' (HFR) is the most common cause for most types of SATD, as it is flexible and often accompanies other causes (namely, the comments having multiple causes). Some causes are almost exclusive to specific types, with few occurrences outside a particular type. These cases are TNW and INT for *Test*, IC for *People*, and UCP for *Architecture*; this was reasonable, as these causes are linked to issues specific to those particular TD types.

5 Discussion & Implications

Our study achieved the best results for SATD detection using the PTMs models, with ALBERT having slightly better performance $F1 = 86.21\%$) overall compared to RoBERTa, followed by CNN (in the deep neural network category), and finally

Max Entropy (in the ML category). This is comparable to the latest work (at the time of writing) in SATD detection for OO languages (Ren et al. 2019). PTMs have never been used in SATD detection for OO languages and they have been shown to perform better than traditional deep neural algorithms and machine learning algorithms for software engineering tasks such as classification (Minaee et al. 2020).

Similarly, our study uncovered seven additional SATD types compared to the latest (at the time of writing) SATD work (Liu et al. 2018). However, some of the SATD types in R have lower F1 scores when compared to OO projects. However, this is the first study uncovering these additional types. Thus, as future work, we will try different search strategies to gather better training samples for those specific types to investigate whether they yield better accuracy.

Lastly, compared to the latest work regarding SATD causes (Mensah et al., 2018), we identified eight additional SATD causes in R. (Mensah et al. 2018) had focused only on code debt and therefore identified only seven causes of SATD (one of which we split into two, yielding eight base causes). Despite our new causes not being investigated in OO projects, they are generic enough that they could be applicable to OO. For instance, code written in OO languages (e.g., Java) has comments that could be classified as 'dead or unused code' or 'warning to developers.' These new causes do not seem to be restricted to R programming. However, additional empirical studies are needed to investigate and confirm these hypotheses.

5.1 Difficulty of detecting SATD types

Previous studies demonstrated that R-package developers do not formally elicit requirements but instead 'decide by themselves' on what to work on next and add comments to plan ahead (German et al. 2013; Pinto et al. 2018). R developers perform ad-hoc elicitation of requirements, taking notes as source code comments, and using that to organize their work, rather than following any development lifecycle (Vidoni 2021b; Pinto et al. 2018). Though this happens in traditional OO development, it is not a widespread practice and influences what scientific developers write as comments (Vidoni 2021b). This is supported by these findings below:

- *Requirements Debt* was more challenging to identify, as the structure, wording and quality of the comments in this type fluctuated considerably. Moreover, because R packages can be used in multiple domains (e.g., bioinformatics, geography, finances, survey processing), the features/requirements documented in those comments also varied considerably, affecting the classification difficulty.
- This behaviour also increased the presence of specific causes for SATD introduction, such as 'Failure to Remember', 'Instructions and Steps', 'Missing or Incomplete Features' and 'Misunderstanding Features'.

Likewise, *Algorithm Debt* was challenging to detect. This debt "corresponds to suboptimal implementations of algorithm logic in deep learning frameworks. Algorithm debt can pull down the performance of a system" (Liu et al. 2020). It is a recently-identified TD type that does not appear in OO software, being exclusive to

scientific and statistical software (Liu et al. 2020; Vidoni 2021b). Therefore, prior works based on OO programming languages have not included any keywords for this type of debt nor attempted to identify it automatically. Our analysis detected that the wording used on these comments varies considerably given the nature and type of algorithms implemented in a specific package. As before, the particular domain (e.g., bioinformatics, geography) affects how comments of this debt are worded. The following two examples showcase the versatility of comments belonging to this type: **`soil depths for naming columns it seems that depth is not explicitly exposed but thickness is`** (referring to a particular algorithm regarding soil depth and thickness), and **`inference type depends on method normal both bootstrap only confidence for now`** (regarding statistical bootstrapping). As a result, a more specific study of *Algorithm Debt* is required to identify the nuances in comments.

5.2 Causes that Introduce SATD

'Failure to Remember' (HFR) comments were left as future to-dos; this is aligned with previous findings regarding using SATD comments to organize requirements (Pinto et al. 2018; Vidoni 2021b). The reasoning behind the cause is different to that of OO programming. This is also related to 'Missing/Incomplete Features' (MIC) and 'Developer Questions' (DQ) for most types of debt, even if its presence is minimal. Even when previous studies demonstrated that about 11% of the comments are commented-out and left clogging files (Vidoni 2021b), a few SATD comments warned about its presence (namely, the category 'Dead or Unused Code' (DC)). This may indicate that R developers are not fully aware of the negative consequences of this practice, but further studies are needed. As discussed in Sect. 4.3, we considered that while a comment existed in the code, it was HFR (as it was written with the purpose of not forgetting to take action). It is also worth noticing that these comments can sometimes become 'obsolete', namely, being left written even when the issues it admitted were already tackled. Though other works have focused on removing obsolete HFR comments (Gao et al. 2021), they only did so with explicit 'to do' comments. This study demonstrated that HFR is one of the most versatile causes that can often combine with other causes, effectively altering the wordings used. As a result, future studies could work on expanding the obsolescence detection, as well as investigating for how long such comments endure.

Several causes were more predominant in specific types, having almost no presence among others. This is reasonable since these causes are intrinsically linked to a particular action, period, or process in the software development lifecycle. For example, 'Tests Not Working' (TNW) and 'Insufficient Testing' both appear almost exclusively in *Test Debt*, as they are related to specific test smells; our detection also supports prior findings in R packages' testing (Vidoni 2021a). Another example is 'Unexpected Performance' (UCP), mainly associated with *Architecture Debt*. The fact that some causes are specific to TD types is expected since these examples are very specific and have less generic reasons, e.g., workaround or to-do.

Finally, many comments were classified into two categories, and only two records of the 1300+ comments were classified into three categories.

- The first one was **`look for a fetchtimestamp method appropriate to this item s fetcher this code is a bit fragile`**, which belonged to *Architecture Debt*, and was considered as a BUG (fragile code was interpreted as not working as expected), WTD (warning developers about steps to take), and WOH (since it was a hack to make the code work).
- The other case was **`to-do need to find a way to denote categories ? error chr cannot assign a category to an app to-do`** in *Defect Debt*, categorized as BUG (because there was an explicit error in the code), LOK (because 'need to find a way' indicates the developer did not know how to do something yet) and DQ (because it was posed as a question for other developers finding this and willing to assist).

Since it was not possible to analyze the comments compared to the related code (as explained in Sect. 3), the comments were found equally suitable to multiple causes and were thus marked as such. Overall, 39 (2.9%) of the sampled comments were classified according to more than one cause. This demonstrates that although some TD occurrences are straightforward and derived from a single cause, a percentage of them has multiple causes. Albeit the number of cases was small, this is a flag for TD management, because to 'repay' a particular debt, all of its causes need to be managed (Freire et al. 2020).

Regardless of these difficulties, our findings can also contribute to enabling future works related to 'on-hold SATD' (Maipradit et al. 2020b, a), since prior automated works only focused on large corporate Java projects and did not typify TD instances but instead worked with a simple binary (TD/nonTD) classification.

5.3 Implications

Our research provides detailed insights about SATD in R packages, but a key implication is demonstrating that using existing SATD results based on OO analysis (mostly limited to statically-typed, large-scale Java or C++ projects) results in an incomplete picture of SATD in R. For example, prior work by Vidoni (2021b) had already demonstrated that SATD in R has a different frequency of types, but also added 11 new textual patterns for its identification–this means that, without those patterns, further automation would have created a considerable amount of false negatives. Our work demonstrates that automating detection based on different patterns is feasible while also obtaining high precision values.

In this study, we compared three groups of techniques to automate SATD detection, using PTMs for the first time and demonstrating that they can detect several TD types, even with low samples. Note that we detected 12 TD types, in contrast to previous works that worked with 2-5 types only. This implies that, without this study, other TD types would remain undetected and thus less studied; our work

also contributes to the broader SATD field by demonstrating automation of other TD types (beyond the traditional Code, Test, Defect, Design and Architecture) is achievable.

Moreover, because PTMs are underutilized in terms of SATD detection, our results can help researchers make informed decisions about the best performing models for future studies. Our findings also contribute to demonstrating the flexibility of PTMs, and their high accuracy, even when working on nuanced domains. This is because the techniques we studied achieve approximately 86 F1-score on identifying SATD in a large imbalanced dataset, and the best technique obtains a micro-average F1-score of above 70% for classifying different types of SATD, which are values above the performance of traditional ML methods used so far.

We found that some types of SATD are harder to detect, and this can be a line of research to develop techniques for detecting these types.The main implication of this is that, although some TD types are quite straightforward and someone explicit on the admission of TD introduction/repayment, others are more nuanced and possibly explicit, presenting a more extensive vocabulary not as easily detected. Therefore, our results can be used to produce tools that not only detect more types of TD (we automatically detected 12, when prior work focused only on 2-5) but also can be translated to a tool to assist computation scientists to improve the robustness of their code. Therefore, our findings are usable and relevant beyond software engineering and software development, potentially assisting scientists in other disciplines. With our new types and causes combined with findings of OO SATD studies, the R community can focus on issues that are specific to R, either by developing guidelines for authors to review their code prior to submitting their packages or for organizations such as rOpenSci (Codabux et al. 2021), to train their reviewers which will review packages prior to publishing them.

R developers can use our research results to develop guidelines for others, explaining which types of comments to look for when reviewing the code for submission to the Comprehensive Archive Network (CRAN)[8]. CRAN is the primary archive to submit user-generated R packages or other package peer-review organizations such as BioConductor and rOpenSci. Such a process would minimize and better manage the debt existing in their packages.

Lastly, our dataset is publicly available for replication purposes or complementary studies. This will help advance R research in software engineering as it is currently lacking.

6 Threats to validity

The section elaborates on the threats to the validity of this study and how they were mitigated.

External validity External validity refers to the ability to generalize results. This study was conducted on scientific software, more specifically, R packages.

[8] https://cran.r-project.org/.

Therefore, we cannot extend our results to R scripts or literate programming (i.e., RMarkdown files). However, our results are likely applicable to other scientific packages but probably not to OO or procedural languages. The six models used to study RQ1 and RQ2 can be applied to any text classification task and are not specific to R. It will be useful to study their efficiency in detecting different types of SATD in OO programming languages (out of the scope of this study). However, the new causes described in Sect. 4.3 are generalizable enough to be extended to other languages, regardless of the programming paradigm, but further studies are required to confirm this.

Internal validity This threat refers to the possibility of having unwanted or unanticipated relationships. Our dataset comes from a previous study, and there might be bias in the labelling or wrong labelling. As we apply the classification task, bias might be introduced in the results specific to the test set. To alleviate this problem, we applied the stratified k-fold cross-validation, giving a chance for every sample to be used in the test set once while preserving the distribution of the classes in the train and test sets.

Both datasets, especially the second dataset D_2, which has fewer comments divided into 12 classes, was highly imbalanced. We tried different techniques to balance the datasets as mentioned in Sect. 3.3 and trained models with the balanced data. Then, based on the best result obtained, we chose the balancing technique. This is done explicitly for the ML models to ensure that the best possible results are reported.

Another threat relates to the SVM and LR results, which have lower performance for detecting SATD/non-SATD or SATD types. For these classifiers, different features can be used to train the models: Term-Frequency Inverse Document Frequency (TF-IDF), Count Vectorizer, word embedding (from deep learning models), and n-grams. We trained the models with different features to provide a fair comparison to identify the best-performing one. The results we reported for these classifiers are the best models for our datasets (see Sect. 3.3), effectively mitigating this threat.

Pre-processing the dataset can introduce some threat to the validity of the results. We could not find a consensus in the literature about the pre-processing steps in the SATD detection studies. Since this is rarely reported, and there are variations about the pre-processing, we applied the techniques found in previous studies (da Silva Maldonado et al. 2017). Although lemmatization is used in many text classification works in software engineering (Stanik et al. 2019; Maalej and Nabil 2015; da Silva Maldonado et al. 2017), we applied Max Entropy on the dataset without lemmatization to examine its effects, reporting the results in Sect. 4.2.1. For the PTMs, there is no previous study in SATD detection that uses these models. Therefore, we applied the best practices of NLP to process the data (Liu et al. 2019; Kanade et al. 2020; Lan et al. 2020). Other pre-processing techniques such as stop word removal and lemmatization are reported rarely in the literature (Huang et al. 2018; Mensah et al. 2018). However, experimenting with different pre-processing techniques and assess their effect in the detection process is out of scope of this work. Some researchers also mention it as a future work to improve their models (AlOmar et al. 2022).

The authors who trained the models are aware of ML theories and deep learning and tested them carefully to ensure they are not overfitted. Thus, we do not anticipate a threat related to training the models and reported results. Our findings with balancing the datasets contradict with the findings of Sridharan et al. (2021), so we experimented with building classifiers both for balanced and imbalanced data, following the approach of Wang et al. (2020) for handling imbalanced data in SATD detection.

Lastly, the manual process of identifying the causes of SATD can be biased. To mitigate this threat, two authors separately classified the SATD according to the causes and then discussed any discrepancy to finalize the classification. Moreover, both authors have over ten years of experience as developers, are well-versed in TD, and one of them has over five years of experience in R programming.

Construction validity Construction validity refers to the degree to which a test measures what it claims to be measuring. Some of the SATD types (e.g., *People* and *Versioning*) had a low number of occurrences and can be a potential bias for our findings. However, we alleviated this threat with the techniques mentioned in the internal validity section.

7 Conclusion

We conducted a pioneering study to automatically identify Self-Admitted Technical Debt (SATD) in scientific software, namely, R packages, using comments extracted from 503 R packages, representing over 160k source code comments.

We studied the performance of three types of techniques: machine learning (ML), deep learning (DL), and neural pre-trained models (PTMs) for the automatic detection of SATD, with this being the first study to use PTMs for SATD per-type classification. We successfully automatically identified 12 types of SATD, which represents a considerable improvement given prior works worked with 2-5 types only. Moreover, automatic SATD detection in the context of R packages has not been addressed before in previous studies. We also manually classified comments samples and determined eight new possible causes (a total of 16) for SATD introduction; this represents an improvement from prior works that only identified eight causes.

Therefore, the results of our study can help investigate scientific software systems and mixed-paradigm languages. We found that some types of SATD are easier to detect, and some are harder for all models, which may be related to the varied nature of scientific software. Investigating the reasons can be a future line of research. Additionally, we determined that PTMs are the best classifiers for these tasks, managing excellent results without overfitting even in cases where SATD types had a low number of comments (i.e., small samples).

In the future, to better understand SATD, we plan to investigate its causes more thoroughly, in particular, the rationale and context of the most prominent causes and the comments that pertain to more than one plausible cause. Besides, we will explore the management of SATD, including prioritization and removal to improve the quality of the software. Another line of work includes investigating why certain SATD types are more difficult to detect than others and developing

new techniques to improve their detection, including experimenting with different pre-processing techniques. Lastly, we want to implement a tool to support our automation and test it with computational scientists writing scientific software as part of their research.

Acknowledgements 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, and RGPIN-2019-05175 at the University of British Columbia.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions.

References

Ahalt, S., Band, L., Christopherson, L., et al.: Water science software institute: agile and open source scientific software development. Comput. Sci. Eng. **16**(3), 18–26 (2014). https://doi.org/10.1109/MCSE.2014.5

Ahmad, W., Chakraborty, S., Ray, B., et al.: A transformer-based approach for source code summarization. In: 58th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, Online, pp. 4998–5007,https://doi.org/10.18653/v1/2020.acl-main.449 (2020)

Alfadel, M., Costa, DE., Shihab, E.: Empirical analysis of security vulnerabilities in python packages. In: IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 446–457, (2021) https://doi.org/10.1109/SANER50967.2021.00048

AlOmar, E.A., Christians, B., Busho, M., et al.: Satdbailiff-mining and tracking self-admitted technical debt. Sci. Comput. Program. **213**(102), 693 (2022). https://doi.org/10.1016/j.scico.2021.102693

Arvanitou, E.M., Ampatzoglou, A., Chatzigeorgiou, A., et al.: Software engineering practices for scientific software development: a systematic mapping study. J. Syst. Softw. **172**(110), 848 (2021). https://doi.org/10.1016/j.jss.2020.110848

Arya, D., Wang, W., Guo, J.L., et al.: Analysis and detection of information types of open source software issue discussions. In: 41st International Conference on Software Engineering, pp. 454–464. IEEE/ACM, Canada (2019)

Bavota, G., Russo, B.: A Large-Scale Empirical Study on Self-Admitted Technical Debt. In: 13th International Conference on Mining Software Repositories. ACM, USA, MSR '16, pp. 315–326, (2016) https://doi.org/10.1145/2901739.2901742

Bogart, C., Kästner, C., Herbsleb, J., et al.: How to break an api: Cost negotiation and community values in three software ecosystems. In: 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, FSE 2016, p. 109–120, (2016) https://doi.org/10.1145/2950290.2950325

Chawla, N.V., Bowyer, K.W., Hall, L.O., et al.: Smote: synthetic minority over-sampling technique. J. artif. intell. res. **16**, 321–357 (2002)

Codabux, Z., Williams, B.J., Bradshaw, G.L., et al.: An empirical assessment of technical debt practices in industry. J. Softw. Evol. Process. **29**(10), e1894 (2017). https://doi.org/10.1002/smr.1894

Codabux, Z., Vidoni, M., Fard, FH.: Technical Debt in the Peer-Review Documentation of R Packages: A rOpenSci Case Study. In: IEEE/ACM 18th International Conference on Mining Software Repositories. IEEE, USA, pp. 195–206, (2021)https://doi.org/10.1109/MSR52588.2021.00032

Cruz, D., Santana, A., Figueiredo, E.: Detecting bad smells with machine learning algorithms: An empirical study. In: 3rd International Conference on Technical Debt. ACM, USA, TechDebt '20, p. 31–40, (2020) https://doi.org/10.1145/3387906.3388618

Cui, Y., Jia, M., Lin, T., et al.: Class-balanced loss based on effective number of samples. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition. IEEE, Long Beach, CA, USA, pp. 9260–9269, (2019)https://doi.org/10.1109/CVPR.2019.00949

Cunha, WS., Armijo, GA., de Camargo, VV.: Investigating non-usually employed features in the identification of architectural smells: A machine learning-based approach. In: 14th Brazilian Symposium on Software Components, Architectures, and Reuse. ACM, USA, SBCARS '20, p. 21–30, (2020)https://doi.org/10.1145/3425269.3425281

da Silva, M.E., Shihab, E., Tsantalis, N.: Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt. IEEE Trans. Software Eng. **43**(11), 1044–1062 (2017). https://doi.org/10.1109/TSE.2017.2654244

Decan, A., Mens, T., Claes, M.: On the topology of package dependency networks: A comparison of three programming language ecosystems. In: Proccedings of the 10th European Conference on Software Architecture Workshops. Association for Computing Machinery, New York, NY, USA, ECSAW '16, (2016) https://doi.org/10.1145/2993412.3003382

Decan, A., Mens, T., Claes, M., et al.: When github meets cran: an analysis of inter-repository package dependency problems. In: 23rd International Conference on Software Analysis, Evolution, and Reengineering, vol. 1, pp. 493–504. IEEE, Suita, Japan (2016)

Devlin, J., Chang, MW., Lee, K., et al.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: Proceedings of NAACL-HLT 2019. ACL, Minneapolis, Minnesota, p. 4171–4186(2019)

Fan, A., Lewis, M., Dauphin, Y.: Hierarchical neural story generation. In: 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics, Melbourne, Australia, pp. 889–898, https://doi.org/10.18653/v1/P18-1082(2018)

Flisar, J., Podgorelec, V.: Enhanced Feature Selection Using Word Embeddings for Self-Admitted Technical Debt Identification. In: 44th Euromicro Conference on Software Engineering and Advanced Applications. IEEE, Prague, Czech Republic, pp. 230–233, (2018)https://doi.org/10.1109/SEAA.2018.00045

Flisar, J., Podgorelec, V.: Identification of Self-Admitted Technical Debt Using Enhanced Feature Selection Based on Word Embedding. IEEE Access 7:106,475–106,494. https://doi.org/10.1109/ACCESS.2019.2933318(2019)

Forman, G., Scholz, M.: Apples-to-apples in cross-validation studies: pitfalls in classifier performance measurement. SIGKDD. Explor. Newsl. **12**(1), 49–57 (2010). https://doi.org/10.1145/1882471.1882479

Fowler, M.: Technical debt quadrant. https://martinfowler.com/bliki/TechnicalDebtQuadrant.html (2009)

Freire, S., Rios, N., Gutierrez, B., et al.: Surveying Software Practitioners on Technical Debt Payment Practices and Reasons for Not Paying off Debt Items. In: Proceedings of the Evaluation and Assessment in Software Engineering. ACM, USA, EASE '20, p. 210–219, https://doi.org/10.1145/3383219.3383241 (2020)

Fucci, G., Cassee, N., Zampetti, F., et al.: Waiting around or job half-done? sentiment in self-admitted technical debt. In: 18th International Conference on Mining Software Repositories. IEEE, Madrid, Spain, pp. 403–414, https://doi.org/10.1109/MSR52588.2021.00052 (2021)

Gao, Z., Xia, X., Lo, D., et al.: Automating the Removal of Obsolete TODO Comments. In: 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, USA, ESEC/FSE 2021, p. 218–229, (2021)https://doi.org/10.1145/3468264.3468553

German, DM., Adams, B., Hassan, AE.: The Evolution of the R Software Ecosystem. In: 17th European Conference on Software Maintenance and Reengineering. IEEE, Genova, Italy, pp. 243–252, (2013) https://doi.org/10.1109/CSMR.2013.33, iSSN: 1534-5351

Haibo He, Y.M.: Imbalanced Learning: Foundations, Algorithms, and Applications. Wiley-IEEE Press, USA (2013)

Hannay, JE., MacLeod, C., Singer, J., et al.: How Do Scientists Develop and Use Scientific Software? In: ICSE Workshop on Software Engineering for Computational Science and Engineering. IEEE, Vancouver, Canada, pp. 1–8, (2009) https://doi.org/10.1109/SECSE.2009.5069155

Howison, J., Deelman, E., McLennan, M.J., et al.: Understanding the scientific software ecosystem and its impact: current and future measures. Res. Eval. **24**(4), 454–470 (2015). https://doi.org/10.1093/reseval/rvv014

Huang, Q., Shihab, E., Xia, X., et al.: Identifying Self-Admitted Technical Debt in Open-Source Projects Using Text Mining. Empir. Softw. Eng. **23**(1), 418–451 (2018)

James, G., Witten, D., Hastie, T., et al.: An Introduction to Statistical Learning, vol. 112. Springer, USA (2013)

Jiang, M., Wu, J., Shi, X., et al.: Transformer based memory network for sentiment analysis of web comments. IEEE Access. **7**(179), 942–179953 (2019)

Kanade, A., Maniatis, P., Balakrishnan, G., et al.: Learning and evaluating contextual embedding of source code. In: International Conference on Machine Learning, pp. 5110–5121. PMLR, USA (2020)

Kaur, A., Dhiman, G.: A review on search-based tools and techniques to identify bad code smells in object-oriented systems. In: Yadav, N., Yadav, A., Bansal, J.C., et al. (eds.) Harmony Search and Nature Inspired Optimization Algorithms, pp. 909–921. Springer, Singapore (2019)

Kaur, A., Jain, S., Goel, S.: A support vector machine based approach for code smell detection. In: International Conference on Machine Learning and Data Science, pp. 9–14. IEEE, Noida, India (2017)

Krishnaveni, S., Vigneshwar, P., Kishore, S., et al.: Anomaly-based intrusion detection system using support vector machine. In: Artificial Intelligence and Evolutionary Computations in Engineering Systems, pp. 723–731. Springer, Singapore (2020)

Lan, Z., Chen, M., Goodman, S., et al.: ALBERT: a lite BERT for self-supervised learning of language representations. In: International Conference on Learning Representations, pp. 1–13. ICLR, Addis Ababa, Ethiopia (2020)

Li, Y., Soliman, M., Avgeriou, P.: Identification and remediation of self-admitted technical debt in issue trackers. In: 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, Portoroz, Slovenia, pp. 495–503,(2020) https://doi.org/10.1109/SEAA51224.2020.00083

Lin, T., Goyal, P., Girshick, R., et al.: Focal loss for dense object detection. IEEE Trans. Pattern Anal. Mach. Intell. **42**(2), 318–327 (2020). https://doi.org/10.1109/TPAMI.2018.2858826

Liu J, Huang, Q., Xia, X., et al.: Is Using Deep Learning Frameworks Free? Characterizing Technical Debt in Deep Learning Frameworks. In: 42nd International Conference on Software Engineering: Software Engineering in Society. ACM, USA, ICSE-SEIS '20, p. 1–10, (2020)https://doi.org/10.1145/3377815.3381377

Liu, Y., Ott, M., Goyal, N., et al.: Roberta: A Robustly Optimized BERT Pretraining Approach. arXiv preprint arXiv:1907.11692 1(1):1–13 (2019)

Liu, Z., Huang, Q., Xia, X., et al.: SATD Detector: A Text-Mining-Based Self-Admitted Technical Debt Detection Tool. In: 40th International Conference on Software Engineering: Companion Proceedings. ACM, USA, ICSE '18, p. 9–12, https://doi.org/10.1145/3183440.3183478 (2018)

Maalej, W., Nabil, H.: Bug report, feature request, or simply praise? on automatically classifying app reviews. In: IEEE 23rd International RE. IEEE, Canada, pp. 116–125, https://doi.org/10.1109/RE.2015.7320414 (2015)

Maipradit, R., Lin, B., Nagy, C., et al.: Automated identification of on-hold self-admitted technical debt. In: 20th International Working Conference on Source Code Analysis and Manipulation, pp. 54–64. IEEE, Adelaide, SA, Australia (2020)

Maipradit, R., Treude, C., Hata, H., et al.: Wait for it: identifying "on-hold'' self-admitted technical debt. Empir. Softw. Eng. **25**(5), 3770–3798 (2020)

Maldonado, E., Shihab, E.: Detecting and quantifying different types of self-admitted technical debt. In: 7th International Workshop on Managing Technical Debt. IEEE, Bremen, Germany, pp. 9–15, https://doi.org/10.1109/MTD.2015.7332619 (2015)

Manning, C., Klein, D.: Optimization, maxent models, and conditional estimation without magic. In: Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology. Association for Computational Linguistics, USA, NAACL-Tutorials '03, p 8, (2003) https://doi.org/10.3115/1075168.1075176

McHugh, M.L.: Interrater reliability: the kappa statistic. Biochemia. medica. Biochemia. medica. **22**(3), 276–282 (2012)

Mensah, S., Keung, J., Bosu, MF., et al.: Rework effort estimation of self-admitted technical debt. In: First International Workshop on Technical Debt Analytics. CEUR-WS, Hamilton, New Zealand, p 6 (2016)

Mensah, S., Keung, J., Svajlenko, J., et al.: On the Value of a Prioritization Scheme for Resolving Self-Admitted Technical Debt. J. Syst. Softw. **135**, 37–54 (2018). https://doi.org/10.1016/j.jss.2017.09.026

Milewicz, R., Pinto, G., Rodeghero, P.: Characterizing the roles of contributors in open-source scientific software projects. In: IEEE/ACM 16th International Conference on Mining Software Repositories, pp. 421–432, https://doi.org/10.1109/MSR.2019.00069 (2019)

Minaee, S., Kalchbrenner, N., Cambria, E., et al.: Deep learning based text classification: A comprehensive review. arXiv preprint arXiv:2004.03705 0(0):1–13 (2020)

Mora-Cantallops, M., Sicilia, M.A., Garcia-Barriocanal, E., et al.: Evolution and prospects of the comprehensive r archive network (cran) package ecosystem. Journal of Software: Evolution and Process **32**(11), e2270 (2020). https://doi.org/10.1002/smr.2270, e2270 smr.2270

Mora-Cantallops, M., Sánchez-Alonso, S., García-Barriocanal, E.: A complex network analysis of the comprehensive r archive network (cran) package ecosystem. J. Syst. Softw. **170**(110), 744 (2020). https://doi.org/10.1016/j.jss.2020.110744

Morandat, F., Hill, B., Osvald, L., et al.: Evaluating the design of the R language. In: Noble, J. (ed.) ECOOP 2012 - Object-Oriented Programming, pp. 104–131. Springer, Berlin (2012)

Mukherjee, S., Almanza, A., Rubio-González, C.: Fixing Dependency Errors for Python Build Reproducibility, pp. 439–451. Association for Computing Machinery, New York, NY, USA (2021)

Naseem, U., Razzak, I., Musial, K., et al.: Transformer based deep intelligent contextual embedding for twitter sentiment analysis. Futur. Gener. Comput. Syst. **113**, 58–69 (2020)

Novielli, N., Girardi, D., Lanubile, F.: A Benchmark Study on Sentiment Analysis for Software Engineering Research. In: 15th International Conference on MSR. ACM, USA, MSR '18, p. 364–375, https://doi.org/10.1145/3196398.3196403 (2018)

Phan, TH., Yamamoto, K.: Resolving class imbalance in object detection with weighted cross entropy losses. arXiv e-prints 1(1):1–13 (2020)

Pinto, G., Wiese, I., Dias, LF.: How do scientists develop scientific software? an external replication. In: 25th International Conference on Software Analysis, Evolution and Reengineering. IEEE, Campobasso, Italy, pp. 582–591, https://doi.org/10.1109/SANER.2018.8330263 (2018)

Potdar, A., Shihab, E.: An exploratory study on self-admitted technical debt. In: International Conference on Software Maintenance and Evolution. IEEE, Victoria, Canada, pp. 91–100, https://doi.org/10.1109/ICSME.2014.31 (2014)

Rantala, L., Mäntylä, M.: Predicting technical debt from commit contents: reproduction and extension with automated feature selection. Softw. Qual. J. **28**(4), 1551–1579 (2020). https://doi.org/10.1007/s11219-020-09520-3

Ren, X., Xing, Z., Xia, X., et al.: Neural network-based detection of self-admitted technical debt: from performance to explainability. ACM Trans. Softw. Eng. Methodol. **28**(3), 1–45 (2019)

Robbes, R., Janes, A.: Leveraging small software engineering data sets with pre-trained neural networks. In: 41st International Conference on Software Engineering: New Ideas and Emerging Results. IEEE/ACM, Montreal,Canada, pp. 29–32, https://doi.org/10.1109/ICSE-NIER.2019.00016 (2019)

Santos, RM., Santos, IM., Júnior, MCR., et al.: Long term-short memory neural networks and word2vec for self-admitted technical debt detection. In: ICEIS (2). IEEE, Virtual Conference, pp. 157–165 (2020)

Setyawan, M.Y.H., Awangga, R.M., Efendi, S.R.: Comparison of multinomial naive bayes algorithm and logistic regression for intent classification in chatbot. In: International Conference on Applied Engineering, pp. 1–5. IEEE, Batam, Indonesia (2018)

Siavvas, M., Tsoukalas, D., Jankovic, M., et al.: Technical debt as an indicator of software security risk: a machine learning approach for software development enterprises. Enterprise Information Systems , 1–43 (2020). https://doi.org/10.1080/17517575.2020.1824017

Sierra, G., Shihab, E., Kamei, Y.: A survey of Self-Admitted Technical Debt. J. Syst. Softw. **152**, 70–82 (2019). https://doi.org/10.1016/j.jss.2019.02.056

Sridharan, M., Mantyla, M., Rantala, L., et al.: Data balancing improves self-admitted technical debt detection. In: 18th International Conference on Mining Software Repositories. IEEE Computer Society, USA, pp. 358–368, https://doi.org/10.1109/MSR52588.2021.00048 (2021)

Stanik, C., Haering, M., Maalej, W.: Classifying multilingual user feedback using traditional machine learning and deep learning. In: IEEE 27th REW. IEEE, Jeju, South Korea, pp 220–226, https://doi.org/10.1109/REW.2019.00046 (2019)

Storer, T.: Bridging the chasm: a survey of software engineering practice in scientific programming. ACM. Comput. Surv. (2017). https://doi.org/10.1145/3084225

TIOBE (2020) TIOBE Index - The Software Quality Company. Online, https://www.tiobe.com/tiobe-index/

Vidoni, M.: Evaluating Unit Testing Practices in R Packages. In: IEEE/ACM 43rd International Conference on Software Engineering, pp 1523–1534, https://doi.org/10.1109/ICSE43902.2021.00136 (2021a)

Vidoni, M.: Self-Admitted Technical Debt in R packages: an exploratory study. In: IEEE/ACM 18th International Conference on Mining Software Repositories. IEEE Computer Society, USA, pp 179–189, https://doi.org/10.1109/MSR52588.2021.00030 (2021b)

Wang, Q., Li, B., Xiao, T., et al.: Learning deep transformer models for machine translation. In: 57th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, Florence, Italy, pp. 1810–1822, https://doi.org/10.18653/v1/P19-1176 (2019)

Wang, X., Liu, J., Li, L., et al.: Detecting and explaining self-admitted technical debts with attention-based neural networks. In: 35th International Conference on Automated Software Engineering, pp. 871–882. IEEE, Melbourne, Australia (2020)

Wattanakriengkrai, S., Maipradit, R., Hata, H., et al.: Identifying design and requirement self-admitted technical debt using N-gram IDF. In: 9th International Workshop on Empirical Software Engineering in Practice (IWESEP), pp. 7–12. IEEE, Nara, Japan (2018)

Wehaibi, S., Shihab, E., Guerrouj, L.: Examining the impact of self-admitted technical debt on software quality. In: 23rd International Conference on Software Analysis, Evolution, and Reengineering, vol. 1, pp. 179–188. IEEE, Osaka, Japan (2016)

Whitworth, B., Ahmad, A., Soegaard, M., et al.: Encyclopedia of Human Computer Interaction. Interaction Design Foundation, USA (2006)

Xavier, L., Ferreira, F., Brito, R., et al.: Beyond the Code: Mining Self-Admitted Technical Debt in Issue Tracker Systems, pp. 137–146. Association for Computing Machinery, New York, NY, USA (2020)

Yan, M., Xia, X., Shihab, E., et al.: Automating change-level self-admitted technical debt determination. IEEE Trans. Softw. Eng. **45**(12), 1211–1229 (2018). https://doi.org/10.1109/TSE.2018.2831232

You, Y., Jia, W., Liu, T., et al.: Improving abstractive document summarization with salient information modeling. In: 57th Annual Meeting of the Association for Computational Linguistics, pp. 2132–2141. ACL, Florence, Italy (2019)

Zampetti, F., Serebrenik, A., Di Penta, M.: Was self-admitted technical debt removal a real removal? an in-depth perspective. In: 15th International Conference on Mining Software Repositories, pp. 526–536. IEEE, Gothenburg, Sweden (2018)

Zampetti, F., Serebrenik, A., Di Penta, M.: Automatically learning patterns for self-admitted technical debt removal. In: 27th International Conference on Software Analysis, pp. 355–366. Evolution and Reengineering. IEEE, London, ON, Canada (2020)

Zanella, G., Liu, CZ.: A Social Network Perspective on the Success of Open Source Software: The Case of R Packages. In: Hawaii International Conference on System Sciences. Scholar Space, Hawaii, pp. 471–480, https://doi.org/10.24251/HICSS.2020.058 (2020)

Zhang, T., Xu, B., Thung, F., et al.: Sentiment analysis for software engineering: how far can pre-trained transformer models go? In: International Conference on Software Maintenance and Evolution. IEEE, Adelaide, Australia, pp 70–80, https://doi.org/10.1109/ICSME46990.2020.00017 (2020)

Zhang, Z., Sabuncu, MR.: Generalized cross entropy loss for training deep neural networks with noisy labels. In: 32nd International Conference on Neural Information Processing Systems. Curran Associates Inc., USA, NIPS'18, p 8792–8802 (2018)

Authors and Affiliations

Rishab Sharma[1] · Ramin Shahbazi[1] · Fatemeh H. Fard[1] · Zadia Codabux[2] · Melina Vidoni[3]

Fatemeh H. Fard
fatemeh.fard@ubc.ca

Zadia Codabux
zadiacodabux@ieee.org

[1] Department of Computer Science, University of British Columbia, Kelowna, Canada

[2] Department of Computer Science, University of Saskatchewan, Saskatoon, Canada

[3] CECS School of Computing, Australian National University, Canberra, Australia