The Journal of Systems & Software 188 (2022) 111265

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Understanding Roxygen package documentation in R[☆]

Melina Vidoni[1,*]

Australian National University, CESC School of Computing, Australia

ARTICLE INFO

Article history:
Received 15 June 2021
Received in revised form 6 January 2022
Accepted 8 February 2022
Available online 15 February 2022

Keywords:
Software engineering
Roxygen
Api documentation
Technical debt
Mining software repositories
Developers' survey

ABSTRACT

R is a package-based programming ecosystem that provides an easy way to install third-party code, datasets, and examples. Thus, R developers rely heavily on the documentation of the packages they import to use them correctly and accurately. This documentation is often written using Roxygen, equivalent to Java's well-known Javadoc. This two-part study provides the first analysis in this area. First, 379 systematically-selected, open-source R packages were mined and analysed to address the quality of their documentation in terms of presence, distribution, and completeness to identify potential sources of documentation debt of *technical debt* that describes problems in the documentation. Second, a survey addressed how R package developers perceive documentation and face its challenges (with a response rate of 10.04%). Results show that *incomplete documentation* is the most common smell, with several cases of *incorrect use* of the Roxygen utilities. Unlike in traditional API documentation, developers do not focus on how behaviour is implemented but on common use cases and parameter documentation. Respondents considered the examples section the most useful, and commonly perceived challenges were unexplained examples, ambiguity, incompleteness and fragmented information.

1. Introduction

R is a programming language for statistical and data science analysis, characterised by being dynamically typed, vectorised, lazy and side-effecting, fostering functional and interactive programming, and providing core object-oriented programming (OOP) features (Turcotte and Vitek, 2019). R is a package-based ecosystem and offers an easy way to install third-party code, datasets, tests, documentation and examples (Hornik, 2012). The main R distribution installs a few 'base' and recommended packages. Though CRAN (Comprehensive R Archive Network)[2] distributes R packages, many are developed (and available) on public version control repositories.

R packages are intended for reuse, and its users rely heavily on the documentation made available by the original developers. As a result, poor package documentation can lead to incorrect usage, affecting all packages and scripts that depend on it, becoming a threat to the validity of the analyses written after that Vidoni (2021a). Therefore, it is essential to assess the quality of package documentation. Although quality packages/library documentation is considered relevant by academics and practitioners alike, research has focused on traditional OOP languages.

Technical Debt (TD) is a metaphor used to describe the situation where long-term software code quality is traded for a quick, short-term solution —it implies future costs of maintenance, adaptability and revisions (Tracz, 2015). *Documentation Debt* (DD) is the dimension of TD which occurs due to poor decisions taken regarding the documentation of a system (Rios et al., 2020). Prominent smells are: not existing in a project, incomplete, or outdated (Alves et al., 2014).

Although DD was explored in other domains (Tan, 2015; Head et al., 2018; Stulova et al., 2020), it has not been addressed in R so far. Despite the R community's growth, there is (to the author's knowledge) no research investigating how developers document packages in practice. This study is needed to understand how this is approached, which are the common smells, and what challenges developers face when consulting package documentation.

This investigation addresses this need through a two-part empirical study. Though there are several documentation systems available for R, Roxygen (Wickham and Grolemund, 2017) presents several similarities to Java's Javadoc since the latter was used as inspiration. Considering that Javadoc has been widely studied in the academic literature, a decision was made to explore Roxygen first. The initial part analysed 379 systematically-selected, open-source R packages hosted on GitHub; the dataset included both popular and newer packages, maintained since

☆ Editor: Martin Pinzger.
* Correspondence to: Australian National University, School of Computing, 108 North Road, Canberra ACT 2600, Australia.
E-mail address: melina.vidoni@anu.edu.au.
URL: https://melvidoni.rbind.io/.
1 Researcher.
2 https://cran.r-project.org/.

2019. The second part addressed how R developers perceive documentation and involved the completion and return of an anonymous online survey (response rate close to 10%).

The main contributions of this study are as follows:

- This is the first study conducted to understand package documentation culture among R package developers' community. It analyses the extent to which they are documented by mining open, public repositories hosted in GitHub and performing several analysis.
- It uncovered that R packages suffer from DTD. Many sections of the Roxygen documentation are *incomplete* (in particular, references, returns, and parameters), with multiple cases indicating an *incorrect use* of Roxygen's capabilities (i.e., family tags and topics, examples, code ownership and returns). There are indications that *outdated documentation* may be a problem, but this was out-of-scope.
- It surveys developers to understand their perspective regarding documentation and the challenges they face when using and writing Roxygen.
- Survey results indicate that R developers prioritise different aspects of the documentation (compared to traditional OO development), focusing on parameters, returns and examples, considering the latter the most helpful section. As perceived by the developers, common challenges are unexplained examples, ambiguity, incompleteness, and fragmented information.

This paper is organised as follows. Section 2 discusses related works. Section 3 explains the study's methodology, including research questions, data collection and extraction, and survey design. Results are presented, per question in Section 4. After that, Section 5 discusses interesting points and describes threats to this study's validity. Section 6 concludes the paper and mentions future lines of work.

2. Related works

Documentation in other languages. Several studies examined API (Application Programming Interfaces) documentation, in terms of taxonomies and quality alike (Tan, 2015). For example, Maalej and Robillard (2013) conducted a large-scale manual exploration of the API documentation of two object-oriented languages (OO, namely Java and .NET) to determine taxonomy patterns of knowledge. Tan et al. (2012) completed a first approach to semantically parse Javadoc tags, while Blasi et al. (2017) performed a semantic analysis on them to define code assertions automatically. Stulova et al. (2020) provided a proof-of-concept tool that allows detecting outdated documentation, and Head et al. (2018) investigated what is missing from C++ documentation and whether it was fixed.

MSRs and Surveys. Two-part studies combining mining software repositories (MSR) and developer surveys are common in software engineering research. It was used to assess the popularity of open-source repositories in GitHub (stars and watches) (Borges and Valente, 2018), as well as to evaluate the reasoning behind forking repositories (Jiang et al., 2017). Other authors used it to assess the quality (in terms of smells) of external contributions in GitHub projects (Lu et al., 2018). Another study used two sampling surveys when developers visited the code, combining it with developers and maintainers' interviews to identify API documentation elements that need improvement (Head et al., 2018). Another work investigated ten types of documentation problems that manifest in practice, using a developers' survey (Uddin and Robillard, 2015).

Documentation in R packages. To the author's knowledge, few studies have been conducted to assess how R packages are documented. A mixed-methods MSR investigated self-admitted technical debt in source code comments, determining that about 3% of the comments indicate debt; though developers add it as self-reminders, they are seldom keen on repaying that debt (Vidoni, 2021b). Another MSR investigated the implicit discussion of TD smells during the peer-review of R packages in rOpenSci, determining that although Roxygen and documentation are important for reviewers, developers do not prioritise them as much (Codabux et al., 2021). However, none of these researched Roxygen with the goals posed in this paper.

3. Methodology

This section presents the methodology for this study. This includes the research questions investigated, the MSR to collect data, survey design, and distribution.

3.1. Research questions

This study aims to understand how R packages are documented using Roxygen, to define current issues and challenges. This leads to the following research questions (RQs):

- **RQ1. How many packages use Roxygen documentation?** Related to the smell of *nonexistent documentation* (Alves et al., 2014), it aims to understand if packages are documented or not. It includes the documentation length and if it is compiled and publicly available (as the browsable version, besides inside the IDE).
- **RQ2. How is the Roxygen documentation distributed?** Also aligned with *non-existent documentation* (Alves et al., 2014), this question investigates the presence of Roxygen documentation regarding its distribution on exported/internal functions (equivalent to public and private functions in OOP), functions/methods/classes (for regular packages or those with OOP), as well as internal reuse (by redirecting to topic documentation).
- **RQ3. How complete is the documentation?** This investigates the *incomplete documentation* smell (Alves et al., 2014), by analysing how complete the documentation is. Focused only on functions, to narrow the scope, its goal its to evaluate what tags are being used, the length of the information, and (in some cases) its alignment to the code.
- **RQ4. What challenges do developers face with Roxygen documentation?** Part of the MSR involved collecting public email addresses of the developers; these are disclosed in each package's `description` file. A structured, anonymised survey was sent to them online–the questions aimed to understand their subjective perception of documentation and the challenges they face when using it.

3.2. Repository selection

The mined repositories were selected following the systematic process outlined by Kalliamvakou et al. (2014).

First, inclusion and exclusion criteria were defined to determine which packages should be considered. In summary, the **included R packages** had to be public, open-source repositories written in R, with a basic structure (as defined by Wickham and Grolemund, 2017); packages that provided minimal R code but wrapped other languages were also allowed. **Excluded** packages were forks, written in a natural language other than English, created before 2010 or with no commits after 2019. Personal, deprecated, archived, or not maintained packages were also disregarded; this included books, data packages or collections of other packages unified into a single one.

GitHub's advanced search.[3] was used to filter most of the exclusion criteria through the provided form. Using a 'best match' sorting, the following string was applied: *package NOT personal NOT archived NOT superseded created:>2010-01-01 pushed:>2019-01-01 language:R* This produced 22 308 results, i.e. repositories, as of November 2020. However, regardless of how specific the query terms may be, it is possible that some packages (i.e., 'false positives') were not properly filtered. Moreover, the number of packages to study had to be kept at a 'manageable' level (namely, a representative number that could provide generalisable results in an acceptable time-frame).

As a result, a basic sample calculation (95% of confidence, with 5% of error) was used to determine how many packages should be selected. For a search of 22 308 repositories, the sample size was 378. The search was completed using GitHub's 'best match' sorting,[4] which "combines multiple factors to boost the most relevant item to the top of the result list"; however, there is no clearly defined algorithm for it, and its result (just like any other of GitHub searches) will change over time, as the repositories continue to evolve.

After obtaining the list, the packages were manually inspected in the order of the results. In other words, the author analysed the returned list of packages from the top of GitHub search results and proceeded until surpassing the sample size of 378 valid packages; while doing so, 53 packages were analysed but discarded per the exclusion criteria. Table 1 summarises how many packages were excluded per criteria. As a result, only $379 + 53 = 432$ packages were actually explored (namely, sample size + excluded). **The final sample consisted of 379 R packages that fit the inclusion criteria**.

Table 1
Number of repositories excluded by exclusion criteria during the manual filtering process.

Exclusion criterion	# Repos
Collections of packages	6
Packages as course material	3
Book-related packages	2
Data packages	18
Websites or ShinyApps	1
Tutorial packages	10
Utilities (e.g. badges, dependencies, etc.)	12
Incorrect package structure	0
Superseded (not filtered)	1

3.3. Data extraction

The GitHub URLs of the selected R packages were kept in a CSV file. An R script was used to download every selected repository and extract the Roxygen comments automatically.

Note that, like with other comment-based documentations (namely, Javadocs) a package should (ideally) have multiple comments blocks, covering each function, object and topic. Likewise, as with Javadoc or other languages, Roxygen documentation *should* 'cover' the elements of every function/class/object being documented; for instance, for a function it should detail every parameter, return type, errors and exceptions, and provide a brief description of the overall behaviour. Roxygen provides 'tags' (equivalent to Javadoc's `@tag`) to indicate that a 'segment' of the documentation refers to a given attribute of a specific function. The full list of tags is available online.[5] Therefore, to analyse these 'segments' (namely, the pieces that are 'part'), they had to be divided but remain associated to the Roxygen block they belonged to.

To do this, the following process was completed automatically for each downloaded package using an R script, that performed the following steps:

1. Read all R files located in the `R` folder (equivalent to Java's `src` folder). Test files where ignored.
2. For each file, it extracted all the lines of the Roxygen comment by detecting the starting `#'` structure using a regular expression. This step generated a dataset that included: package name, file name, comment ID number, start and end line, current line number, roxygen text, function signature.
3. Each extracted comment was divided into *segments*, namely, groups of consecutive lines that belong to a tag, as explained above. An auto-incremental 'segment id' field was added to the above dataset (in spreadsheet format). Multiline comments were kept together by continuing reading lines until (a) there was a blank line, or (b) a new tag started. Thus, every row of this dataset had a key composed of package name, auto-incremental comment number, and auto-incremental segment number.
4. Like in Javadoc, Roxygen tags are identified with an `@` symbol. These were extracted to a new column of the dataset using regular expressions. The full list of tags is available online.[6]

For example, Fig. 1 shows on the left a small Roxygen block, and at the right its partition into multiple segments. As can be seen, for that single function, the Roxygen documentation accounted for the following segments: a title (always the first sentence), a description (the first paragraph after that), three parameters, one return, and one visibility. Note that, as per point (2) above this was organised into an spreadsheet, but is presented here as a figure for ease of reading. Also note that the Roxygen tags present in this example are `@param` (three of them), one `@return` and one `@export`.

The complete dataset had 8670 Roxygen comments, **with a total of 861 601 lines of Roxygen comments**.

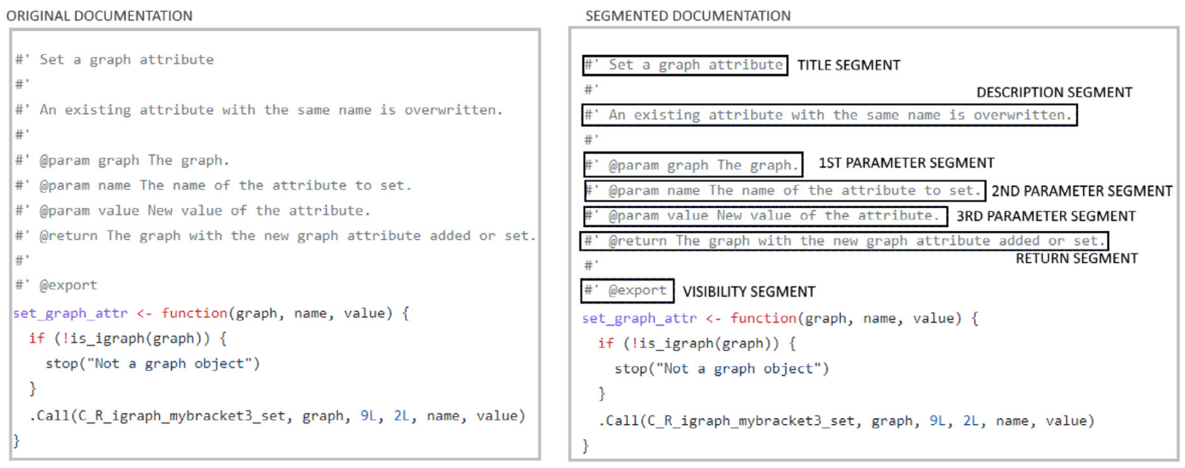
3.4. Developers' survey

This part of the study consisted of an anonymised online survey. R packages list the information about maintainers in the `description` file, including their email address. These emails are publicly shown in CRAN (if the package is there) and in the IDEs (Integrated Development Environment) upon installation. An R script was used to extract this information and remove the duplicates, obtaining 478 unique email addresses; in many cases, one author had participated in several mined packages.

This survey was implemented in Qualtrics, and its distribution tools were used to send the survey to potential candidates. The survey remained open from February–March 2021. Email-related information was removed to ensure the anonymity of respondents. Overall, 36 emails bounced, and 48 complete responses were collected (response rate of 10.04%), which close to the average for software engineering surveys of this type (Singer et al., 2008).

The survey was composed of two sections, demographics (programming experience and packages authored) and Roxygen use. It was based on surveys used in two related works by other authors (Head et al., 2018; Uddin and Robillard, 2015), to compare the responses. The demographics had pre-fixed answers, while

[3] See: https://github.com/search/advanced.
[4] See: https://docs.github.com/en/rest/reference/search.
[5] See: https://roxygen2.r-lib.org/articles/rd.html.
[6] See: https://roxygen2.r-lib.org/articles/rd.html.



```
#' Set a graph attribute
#'
#' An existing attribute with the same name is overwritten.
#'
#' @param graph The graph.
#' @param name The name of the attribute to set.
#' @param value New value of the attribute.
#' @return The graph with the new graph attribute added or set.
#'
#' @export
set_graph_attr <- function(graph, name, value) {
  if (!is_igraph(graph)) {
    stop("Not a graph object")
  }
  .Call(C_R_igraph_mybracket3_set, graph, 9L, 2L, name, value)
}
```

Fig. 1. Example of a Roxygen comment divided into segments. Multiple combinations of segments were possible, according to what the developers documented.

the Roxygen question had a mixture of Likert scales, prefixed answers, and two open-ended questions. In particular, this section included questions about what information is consulted (with a follow-up on behaviour), time spent reading documentation, common problems encountered and their severity, examples of helpful documentation, and other comments

Given its length, the full structure, including questions and answers, is available in Appendix, in Table 13.

This methodology underwent a thorough ethical review and was approved. The packages' names and the extent of the mined comments explored in this study are not shared to preserve the identity of those contacted for the survey and avoid a potential re-identification of the responses.

4. Results

This section reports the result of analysing the documentation of 379 systematically-selected, open-source R packages and the 48 complete survey responses. Results are presented by RQ to be addressed.

4.1. Presence (RQ1)

Documentation for R packages has mutated over time. 'Base' R documentation (namely, the one provided without the installation of any package) has a Latex-like syntax.[7] However, RStudio created `roxygen2`.[8] Roxygen is installed as a package and was inspired by Javadoc to provide an easier, more straightforward way of writing the documentation that quickly became a de-facto standard (Wickham and Grolemund, 2017). Moreover, R's most used IDE (Integrated Development Environment) allows creating a basic skeleton Roxygen from a menu option and checking the function's documentation by invoking `?function_name`.

Therefore, the first analysis of this question evaluated how many packages had every type of documentation. Table 2 summarises the results. Overall, of the 379 included packages, about 9% had no API documentation at all.

Regarding *no documentation*, some packages had a `man` folder for documentation, but its functions had no comments at all, or the folder was empty (10 packages out of 19). Others had *regular comments* (written as `#`) instead of Roxygen's (with `#'`) to write minimal documentation; these were not structured, limited, and sometimes written under a method's signature. The problem is that they are not compiled, and therefore not easily accessible to developers unless they go to the original repository of the package. Another group of packages used regular comments to clarify *code-ownership or contribution*, with no information about the function; these were occasionally located under the signature of the function but are also not standardised.

Table 2
Existence of documentation in R packages.

Situation	# Packages	Compiles
No documentation	19	NO
Regular comments	10	NO
Only authorship	4	NO
Latex documentation	4	YES
Roxygen	342	YES

Finally, only four packages used *Latex-inspired documentation style exclusively*. This type of documentation can be compiled and showcased inside the IDE and has the same functionalities as Roxygen. Nowadays, most books and R resources recommend Roxygen instead of the Latex or Md version[9] (Wickham and Grolemund, 2017; Peng, 2012); given the popularity of `tidyverse` and RStudio solutions among the R community, it is possible to assume that these recommendations affected the use trends of documentation tools. Therefore, these four packages are not included in the following analysis.

From the 379 selected packages, only 342 had Roxygen documentation. Every package had an average of 38 Roxygen documents, with about 4.3 comments per file describing a function.

Findings. Most R packages use Roxygen documentation (about 90% of the sample of this paper). Regarding the remaining 10%, 9% had no documentation at all, and less than 1% was documented using only the original Latex-style.

4.2. Distribution (RQ2)

To continue investigating the smell of *non-existent documentation*, this RQ conducted several analysis to understand how Roxygen comments are distributed. To simplify the reading, these are presented in individual sub-sections. The following analysis were completed on the 342 packages that had Roxygen documentation.

[7] See: https://developer.r-project.org/Rds.html.
[8] See: https://roxygen2.r-lib.org/.
[9] See: https://developer.r-project.org/Rds.html.

4.2.1. General distribution

An R script was used to process the dataset in terms of comments, files, packages and length. This was used to examine the distribution in terms of *comments per package*, *comments per file* and *length in lines of comment*. This is relevant in order to uncover gaps of information, organisation and incomplete documentation, especially in regards to the other points, documented elements and visibility. The results are summarised in packages per quantile in Table 3.

Table 3
Distribution of Roxygen per package and file, and its length.

Measure	25%	50%	75%	100%
Comm. per Package (CPP)	18	38	81.75	8670
Comm. per File (CPF)	7	15	29	680
Length in LOC (LIL)	25	47	93	1111

As can be seen, the average is about 38 comments per package. However, the fourth quantile indicated that about 86 packages had more than 82 comments per package. A mix of automated and manual analysis (namely, selective code inspections to double-check the automated results) identified several elements affecting this:

1. Some packages make extensive use of *topics*. Topics are every element that is documented (e.g., functions or objects), but also groups of elements (namely, `@family`) that encapsulate the documentation of a given group to avoid redundancies (Wickham and Grolemund, 2017). As a result, about a third of this group's comments belong to topics, with many others referencing them.
2. Use of *aliases*–i.e, alternative names for a topic. A common example are the `tidyverse` functions that permit both British and American spelling; for example, `summarise` and `summarize` invoke the same function. A large number of aliases was detected (see Section 4.3).
3. Other studies investigated function clones in R (Claes et al., 2015)–a practice where developers duplicate functions from other packages to reduce the number of dependencies; this is done either by copy-pasting the code of the function or re-exporting it using Roxygen tags. Re-exporting external functions was also detected in this group (more analysis in Section 4.3).
4. Packages that are genuinely large, providing a large number of functions (e.g., those that provide connections to specific APIs), aliases and topics.

The distribution in terms of *comments per file* was also analysed. Due to R's paradigm, there is no limitation regarding the organisation of elements in files like there is in OOP (i.e., all attributes and methods of a class are written in the same file). As a result, mixed advice is shared in the R literature (Wickham and Grolemund, 2017; Peng, 2012), with the most common being: (a) one function per file if it is too long, (b) isolating all auxiliary/internal functions in other files, and (c) keeping the primary function and its auxiliaries in a single file. As a result, the variability in the distribution of CPF can be due to the lack of standardised practices.

Finally, the *length in lines of comment* was automatically calculated. The R script extracted the starting and ending line of every Roxygen comment and then calculated the difference to obtain the length. Note that the length of every line is not fixed and may vary–although CRAN checks recommend lines of 80 characters,[10] not every developer follows this. As a result, though it is safe to assume that packages intended to be published in CRAN have about 80 characters per line, these numbers may not fully represent the comments' size. After this generic data regarding distribution, more specialised analyses were completed.

4.2.2. Documented elements

To quickly identify the generic type, each signature previously extracted was automatically classified with an R script. They were typed as *functions* (methods not defined through OOP), *S3 Methods* (OO methods) and *others*; the latter was sub-typed using regular expressions. Table 4 summarises how many elements are documented.

Table 4
Number of comments per type of documented element.

Type	Sub-type	# Comments	Total #
Functions			26 729
OOP			3118
Others	NULL	2097	7245
	Alias	1581	
	NA	416	
	Deprecated	275	
	Reexports	231	
	Package	53	
	Others	2592	

Package documentation is recommended to use the sentinel `_PACKAGE` when writing the topic,[11] nonetheless many developers still use `NULL`, making it a possible indicator.

Overall, 72% of the documentation belongs to functions, about 8.5% to OOP elements (methods and classes), and the remaining 19.5% to other elements such as aliases, packages, re-exports, topics, and deprecated elements kept for compatibility purposes. These results can support the claim made in different R literature regarding the reduced acceptance of R's OO capabilities (Wickham and Grolemund, 2017; German et al., 2013).

4.2.3. Exported vs. internal

R packages can limit the visibility of an element. Those elements (e.g., functions, aliases, classes) that are 'public' (in an OOP sense) are called *exported*, while those meant to be used only 'privately' are known as *internal* (Wickham and Grolemund, 2017). This visibility can be declared in several ways:

- Using specific Roxygen tags that would automatically mark the function as *exported* in the `namespace` file. These are `@export`, `@method`, `@exportS3Method`, `@exportMethod`. On the other hand, using `@keywords internal` would mark a function as *internal*. Then, a developer can use `devtools` to automatically generate the `namespace` file.
- The `namespace` file can be manually altered. First, it is possible to export all functions that match a specific pattern (namely, using regular expressions) with `exportPattern()`.[12] Second, the tag `export()` or `S3method()` can be added without the need of a Roxygen tag. Though this was a common practice before, nowadays the IDEs warn developers of this practice.

The packages had to be installed to analyse the entirety of a package's elements (beyond those with Roxygen). The first attempt was installing directly from CRAN; for those failing, the author manually explored their Readme files to determine how to install them:

10 See: https://r-pkgs.org/r-cmd-check.html.

11 See: https://cran.r-project.org/web/packages/roxygen2/vignettes/rd.html.

12 See: https://r-pkgs.org/namespace.html.

- Their GitHub repository name did not match the CRAN name. The package was installed from CRAN using the correct name.
- Packages belonged to BioConductor and thus required a different command to be installed.[13]
- Packages were neither on CRAN nor BioConductor and were installed using `remotes::install_github`.

Overall, out of the 342 that contained Roxygen, 83 could not be explored for different reasons. In detail, 41 packages had dependency errors (either they were not available anymore or were outdated versions), 40 packages had build errors and could not be installed, and the remaining 2 failed the installation due to other reasons.

Another R script was used to automatically explore the `namespace` files of the installed packages. Overall, it was based on the family of functions `base::getNamespace`.[14] With this, it was possible to extract all elements that were *exported* or *internal* in the `namespace`; these two categories were exclusive, since an element can be either one of those. However, none of those account for *OOP* which are extracted separately; according to the documentation, if a namespace contains either `S3method()`, `exportClasses()` or `exportMethods()`, then the element is exported.

The script extracted the name of the element in question. That name was matched to the signatures extracted from the R files' text analysis to determine which elements had a Roxygen comment. Table 5 summarises this information.

Table 5
Total number of documented and non-documented elements, obtained by exploring the namespace of the installable packages.

Has Roxygen?	State	# Elements	Total #
NO	Exports	2306	15 775
	OOP	1051	
	Internal	12 418	
YES	Exports	8773	18 874
	OOP	6939	
	Internal	2162	

In terms of *internal* functions, the vast majority is not documented: about 85% does not have Roxygen comments, and only 15% does. This is relevant, as the R operator `:::` (read as 'triple-colon operator') violates this visibility, allowing a developer to reach the internal function without reflection. Though the R documentation considers its use as a 'design mistake'[15] CRAN checks do not contemplate this issue. As a result, the lack of documentation on internal functions can lead to incorrect usages.

There is a very considerable number of elements that are exported but have no Roxygen (about 17.6%), indicating that they are exported by manually altering the `namespace` (either with a pattern or not). Some may be cases of outdated documentation (e.g., the function was previously exported, but then the comment was removed); however, analysing previous documentation versions is out of scope for this paper.

The documented elements were further analysed to list which visibility tags appeared in their documentation. Table 6 indicates how many elements had each type of visibility documentation and the percentage of the total it represents. Even though all elements had Roxygen comments, about a 10.1% did not have visibility tags–thus, it can be concluded that they were either exported by manually updating the `namespace` or that the documentation was outdated.

[13] See: https://www.bioconductor.org/install/.
[14] https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/ns-internals.
[15] https://stat.ethz.ch/R-manual/R-devel/library/base/html/ns-dblcolon.html.

Table 6
Exported elements with Roxygen documentation, and their visibility tags.

Roxygen tag	# Elements	%
@export	14 052	78.6%
@method	1044	5.85%
@keywords	942	5.27%
@exportClass	12	0.07%
@S3method	8	0.04%
@exportMethod	2	0.01%
@exportS3Method	1	0.01%
NA	1813	10.1%

Findings. According to the analyses:
- There is a large variability in the length and distribution of comments, with some of them being disproportionately long (>81 lines).
- According to the namespace exploration, the use of R's OO functionalities is scarce.
- About 85% of the internal functions are not documented.
- About 17.6% of all the exported elements (e.g., functions, aliases, classes) do not have Roxygen documentation.
- About 10.1% of the exported elements with Roxygen documentation do not include any visibility tags.

4.3. Completeness (RQ3)

Incomplete documentation is another smell (Alves et al., 2014), that refers to elements that should be documented but are not. It applies to complete elements or individual parts (e.g., not documenting a function's return but correctly documenting the parameters). This analysis was done by investigating specific sets of Roxygen annotations and comparing them to different elements of the dataset to understand if they are complete or not. As before, this was completed on the 342 packages that had Roxygen documentation.

4.3.1. Visibility declaration

Section 4.2.3 analysed some of the used to declare a method's visibility. As most results and information were stated there, this section provides complementary data. Thus, consider that the discrepancy in some numbers is because Table 6 only analysed the packages that could be installed, while these results include all packages with Roxygen documentation.

The script found 30 408 comments that had visibility related tags. Out of 18 496 using the basic `@export` (about 61%), about 2392 `@export` changed the name of the function in the export; this is not a bad practice, but worth noting. Moreover, about 8921 comments had the `@keywords internal` tag, purposefully marking the element as private and not exportable.

However, a negative observation is that, as of Roxygen 4.0.0 (dated from 2014-05-02),[16] `@S3method` was labelled as deprecated and later removed in favour of `@export`. Therefore, this tag's presence may indicate cases of *outdated documentation* that is not being maintained. On a positive note, no instances of `@exportPattern` were found. This tag was often used either in families of elements in the package documentation to generate a regex-based exporting, and its use is no longer encouraged.

[16] See: https://www.tidyverse.org/blog/2019/11/roxygen2-7-0-0/.

4.3.2. Dependencies management

Given R's package-based nature, the management of dependencies and networks it generates has been a topic of interest for several years (Plakidas et al., 2017; Decan et al., 2017). For instance, it has been determined that the more dependencies a package has, the less likely it is to have a higher impact (Korkmaz et al., 2019).

The script found 7990 instances of these tags because a function may require several imports. As a result, these belonged to only 294 out of 342 packages, meaning that only 4755 comments out of 37 092 had these tags. This indicates that the remainder may be either manually altering the namespace file or incurring in function clones (Claes et al., 2015). In any case, this represents a smell, as it may reduce the maintainability of packages.

Roxygen has two tags to import dependencies:

- `@import [packageName]` imports a full package, and makes them available without using `::`; however, the package under development becomes larger.
- `@importFrom [packageName] [functionName]` imports specific function(s), limiting what is being downloaded and required. It is considered a better practice in terms of installability and performance, but some developers do not recommend it due to poorer readability (Wickham and Grolemund, 2017). Each of these can import one or several functions of a single package, but multiple tags are needed if several packages are required.

Furthermore, `@import` represented only 18% of the instances. On a positive note, this shows that developers are aware of the package's size and thus are conscious about importing only what they need. However, since the source code of the functions was not explored, it is possible that some cases could be replaced by the more limited `@importFrom`.

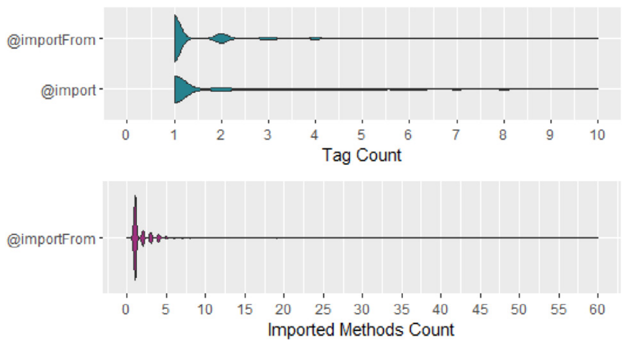


Fig. 2. Distribution of imports and methods in Roxygen comments.

Regarding how many import tags each comment was using, most comments had between 1–3 tags each (see Fig. 2-Top). Regarding the outliers, `@import` capped at ten imports per comment, but `@importFrom` had 17 instances of 11–20 imports and a single outlier with 133 instances of this single tag.

Moreover, a single `@importFrom` can add multiple functions or methods of a single package. Nonetheless, the mean was a single function per package, with 3+ covering 75% of the cases. In terms of outliers, the most outstanding imported 20, 22, 24 and 60 methods from a single package (namely, `stats`) in a single tag each (see Fig. 2-Bottom). Overall, it can be concluded that, if used, the Roxygen-driven importing is correctly applied.

4.3.3. Disclosure of references

Citing and crediting work is an important part of the R community, given that most software is for scientific purposes. Elements that lead to increased citations for an R package are a common topic of research (Korkmaz et al., 2018, 2019; Zanella and Liu, 2020; Li and Yan, 2018; Li et al., 2019). Roxygen provides two tags for this:

- `@seealso` allows to point to other useful resources, either on the web (using the `\link{}` Latex command), or a markdown-flavoured for an internal reference (namely, `[function_name()]`). Originally, it is intended to "cross-link documentation files".[17] It can include multiple links and also an explanation.
- `@references` is stated to be used "to point to published material about the package that users might find helpful"[16].

A script automatically detected 5761 occurrences of these tags (combined), with about 53.5% belonging to `@seealso`. Only 235 packages (out of 342) used either one in 4761 comments, given some comments have multiple instances of each.

There are several problems with the usage of `@seealso` (see Fig. 3). First, about 12% of the instances are empty (hence, *incomplete documentation*), and almost 5% are *incorrectly used*: 3.7% are mentioning another method or object but not providing a working link to it. About 1% are used to link to academic publications or to write text. Second, the 3083 instances of the tag belonged to 3011 comments, meaning that some had more than one occurrence.

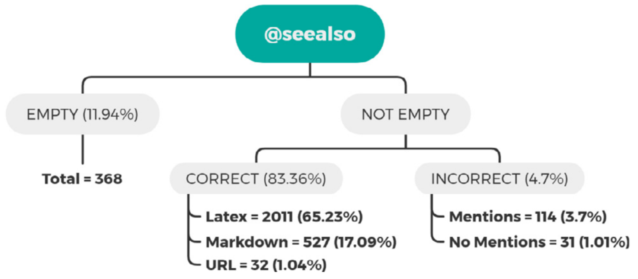


Fig. 3. Detected usages of the `@seealso` Roxygen tag.

Nonetheless, only 266 of those comments were combining this tag with `@family`, which is the recommended best practice[16]. This can be considered *incorrect use*, since the latter is used to cross-reference elements, allowing the reduction of documentation verbosity, simplifying its use. Finally, since including an explanation alongside the links is not required, the messages accompanying them were not analysed.

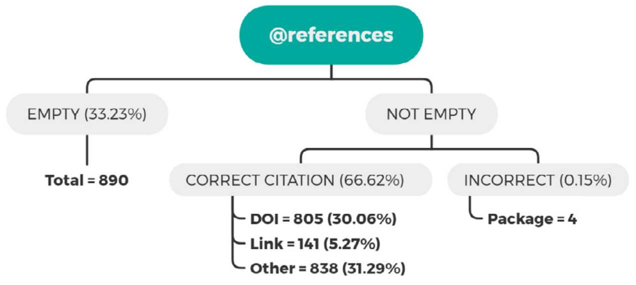


Fig. 4. Detected usages of the `@references` Roxygen tag.

Regarding, `@references` (see Fig. 4), about one third of all the instances are empty (hence, *incomplete documentation*), and about 0.15% are used to link to R packages/methods which is the purpose of `@seealso` (thus, *incorrectly used*). The remaining two-thirds are correct cases that point to an academic publication (with or without a DOI) or any other external link. Not properly

[17] See: https://roxygen2.r-lib.org/articles/rd.html.

referencing the citations being used can be incorrect from an ethics perspective; moreover, it also hinders using an algorithm as the interpretation and implementation cannot be ensured.

4.3.4. Author ownership

'Code ownership' determines rights and duties concerning an object or method, and its effects (especially in open-source software, OSS) have long been studied (Medappa and Srivastava, 2018). Roxygen's `@author` tag is used to indicate a developer and their email address. In OSS, even if the project has a single developer, anybody can contribute; thus, ownership is also considered as crediting the work (Medappa and Srivastava, 2018).

A script detected 3732 usages of the `@author` tag. Only 146 (out of 342) packages used it; this represented 3693 comments (out of 37 092), indicating that some elements had more than one tag for authors. From here, only 2.3% were empty, 36.57% included an email address, and the remaining 61.12% only listed names (occasionally, it added Twitter or GitHub handles, but these were ignored).

This tag has a shallow adoption rate. Though its usage is recommended in different sources,[18] Roxygen's website does not list it in any of the official sources[16], nor the official cheat-sheet. Moreover, exploring the lists of available tags through the methods provided by the package `roxygen2` does list `@author` as one of the existing tags.[19] As a result, it can be concluded that the low spread of this tag is caused by poor documentation of the tool (namely, Roxygen itself) and not due to poor developer practices.

4.3.5. Examples availability

Adding working examples to the R package documentation is considered fundamental in the R community; moreover, CRAN's checks[9] (both automated and manual) enforce that exported methods have a working example. Roxygen has two tags for this purpose:

- `@examples` provide executable R code showing how to use the function in practice; by definition, it must work without errors as it is run automatically as part of the checks performed by CRAN (Wickham and Grolemund, 2017). For this tag, the example must be directly embedded in the specific documentation.
- `@example` allows putting the examples in a source file located in a path relative to the root, inserting them in the documentation (Wickham and Grolemund, 2017). CRAN's checks will still run it, and the compiled version will embed the code, but the source file should only contain the path.

An R script detected 9008 uses of both tags combined, which belonged to 320 packages out of 342–indicating that at least 42 packages do not include examples. Though some business logic may be challenging to exemplify (either by length, performance or other requisites), it is possible to add the Latex command `\\dontrun{...}` to prevent an example from being run. This means that 6.43% packages have *incomplete documentation* in terms of working examples.

In terms of comments, 8940 individual comments had either tag, indicating that some may have two or more uses. Out of the total, `@examples` had 8680 instances, representing 96% of the usage. The remaining cases (328, or about 4%) are for `@example`. Though both are correct, these numbers indicate that developers prefer to write the example directly in the documentation rather than external files. As a result, a more specific classification was conducted to investigate this situation.

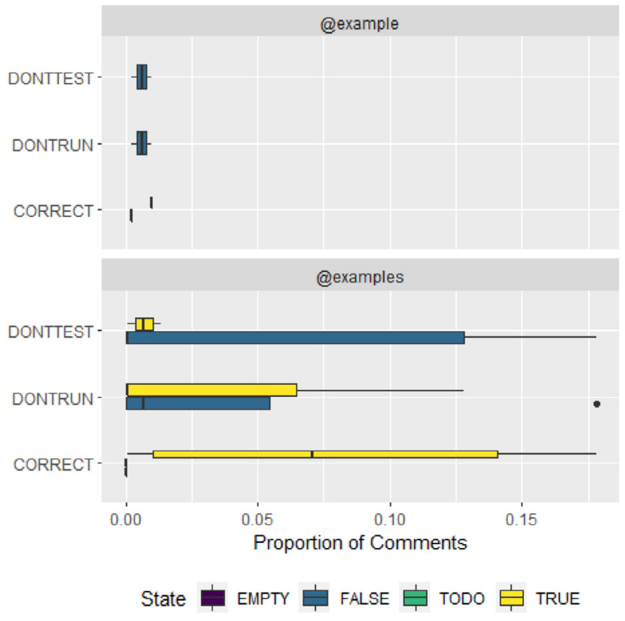


Fig. 5. Analysis of content for Roxygen tags related to examples.

Fig. 5 summarises the content written in this tag. Very few instances were empty, but there were some cases of `todo` annotations, indicating it could be completed in the future. However, when looking only at `@example`, about 17.4% of the instances of this tag were incorrect, with the example directly in the code instead of a path to a source file.

In total, regardless of the tag, almost 38.8% of the instances included the `\dontrun{...}`. Even if the tag is correctly used, the problem is that it prevents CRAN checks from checking that the example works–thus, the developers may be providing an incorrect or outdated example that is no longer useful and could even be misleading. Likewise, only a 4.1% used `\donttest{...}`, which allows an example to be run when compiled but ignored by the CRAN checks. The latter can be related to cases in which the developers purposefully attempt to bypass said checks (Vidoni, 2021b).

In terms of length of the examples, considering only `@examples` (since it can be explored from the raw source file), the median was 32 lines of code per example (which in many cases included regular comments to be showcased on the compiled documentation). The 75th percentile was indicated with 55 lines of code. The shortest examples had a single line, while the longest had 926 lines, being an extreme outlier. Overall, these numbers are positive, as the examples appear to be thoughtfully written.

Finally, the comments the examples belong to were automatically examined for tags related to visibility (see Section 4.2.3). Only those with tags related to exported functions were considered; note that the namespace exploration was not used for this analysis since (as explained before) it was not possible to install all packages. Overall, from the 9008 examples, about 72% belongs to exported (i.e., public) functions, while about 28% was assigned to internal (namely, private) functions. This is positive since R provides the `:::` operator to allow access to those methods or functions.

4.3.6. Return declaration

The tag `@return [description]` describes the output of a function, and it is provided in the Roxygen skeleton automatically constructed in the RStudio IDE (Wickham and Grolemund, 2017).

[18] See: https://kbroman.org/pkg_primer/pages/docs.html.
[19] See: https://github.com/r-lib/roxygen2/issues/792.

An R script identified 8259 usages of this tag present in 310 packages; his means that about 32 packages (almost 9.35%) do not use it at all, leading to *incomplete documentation*.

Regarding *incorrect utilisation*, from all of the usages, four wrote `@returns` (in plural), which is not listed as part of Roxygen's on its official documentation. Moreover, the identified belonged to 8083 comments, indicating that some used the tag multiple times instead of doing an extensive report in a single tag.

Table 7
Distribution of elements that do not document a return.

Sub-type	# Comments	%	Risk
Alias	1493	5.15%	
Package	53	0.18%	Safe
OOP classes	324	1.12%	
Deprecated	275	0.95%	
Others	2351	8.10%	
Reexports	230	0.79%	Mid
NULL	1910	6.58%	
NA	185	0.64%	
OOP methods	2351	8.10%	High
Functions	19 837	63.38%	

Overall, there are 29 009 out of 37 092 (about 78%) that do not document returns. A more detailed analysis was completed using the classification of elements present in Table 7 (in Section 4.2.2). This exploration determined that 76.5% of elements are functions or OOP Methods that should have a return tag, either to indicate its existence or NULL, leading to a severe case of *incomplete documentation*. About 6.5% were elements that do not have returns (classes, package documentation and aliases). In contrast, the remaining 17.07% had a medium risk of being NA or NULL (which can be used to document topics of functions groups), deprecated elements (whose documentation is often wiped out and redirected to the replacement), re-exported function clones, and other types of elements.

Regarding the elements that did have a `@return` tag, about 7.8% was empty, with no description besides the tag (*incomplete documentation*). Overall, the median length was 23 lines of comments per return, which gives enough room for a thoughtful explanation if (following CRAN's checks) the line has about 80 characters; the third quantile indicated 42 lines, and the most extreme outlier had 876 lines of comment in a return.

It is worth noting that R is dynamically-typed, which means that functions and methods do not specify a fixed type of return in their signature (Turcotte and Vitek, 2019; Hornik, 2012). As a result, R can produce several types of returns:

- By visibility of the return, `return(...)` stops the execution and returns the value put into it, while `invisible(...)` is used to return values which can be assigned, but which do not print when they are not assigned.[20] A function that returns visibly in a given path can return invisibly in another.
- By type of output, a function can either store a file on a given location, print a specific element into the console (either information, warnings or errors) or return an object as per the above.
- By type, because under different conditions, multiple types can be returned.

Furthermore, any combination of these three points can be feasibly returned. Therefore, analyses about the completeness of the `@return` tag requiring manually inspecting the code were excluded to limit this manuscript's scope.

20 https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/invisible.

4.3.7. Parameters declaration

The disclosure of parameters and function arguments is essential for API documentation (Head et al., 2018; Blasi et al., 2017). Roxygen provides two different tags for this purpose:

- `@param [argName] [description]` describes the inputs to the function. It can span multiple lines and paragraphs if needed (Wickham and Grolemund, 2017). Roxygen documentations states that "The description should provide a succinct summary of the type of the parameter (e.g. a string, a numeric vector), and if not obvious from the name, what the parameter does" (Wickham and Grolemund, 2017).
- `@inheritParams [sourceFunction]` inherits the documentation from another topic (Wickham and Grolemund, 2017). Thus, how many arguments are documented is not visible on the source file and only on the compiled documentation.

An automated script detected 80 661 instances of the combined tags, with 98% belonging to `@param`. In total, only 322 out of 342 packages documented at least one argument (hence, *incomplete documentation*). It is unrealistic to assume that none of the functions exposed by the missing packages take no arguments. This represented 16 314 comments.

Since arguments are declared in the signature, a semi-automated process determined how many were explained. This was done only on the arguments of functions with a signature, ignoring those belonging to cases such as aliases, re-exports, NULL. Aliases should be redirected with an `@see` or a `@family` topic, rather than specifying a regular documentation body.

Therefore, when exploring function signatures, the script counted how many parameters were on each `@param` tag per function (some authors list two or more arguments on a single tag by splitting them with a comma) listing their names. Then, the script counted how many arguments were part of a function signature, also listing their names. Finally, results were manually revised to ensure that the counts were correct. Lastly, the numbers were compared to determine how many of the signature parameters matched the documentation.

Fig. 6 summarises the first analysis, with results divided according the inclusion of an ellipsis `...` in the signature (y-axis), or the `@inheritParams` in the documentation (columns). Of the total, only 3% of the comments were inheriting parameters,

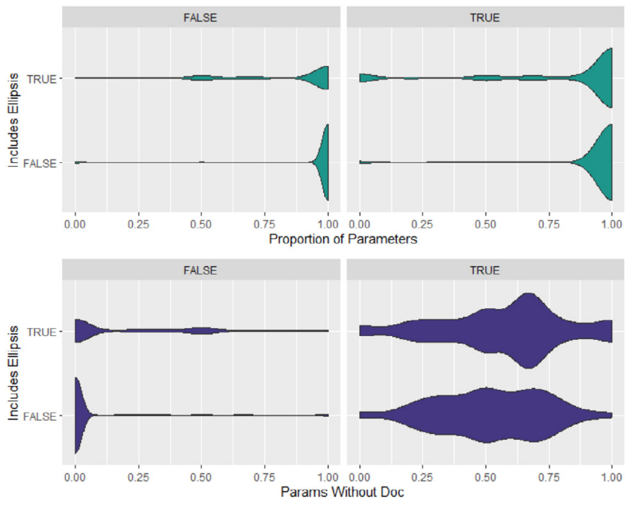


Fig. 6. Proportion of documented parameters that appear in the signature (top), and signature parameters without documentation (bottom). Discriminated by using `...`, and `@inheritParams`.

and about 10.4% had ellipsis arguments. This figure presents two different proportions:

- The top row (deep green) indicates the proportion of documented arguments that appear on the signature. Therefore, values <1 indicate excessive documentation–namely, they are documenting parameters that do not exist in the signature. This can be a cause of *outdated documentation*, but in cases where there is an ellipsis, it may be a *code smell* caused by misusing the variable argument.
- The bottom row (deep purple) indicates the proportion of arguments that appear in the signature but that are not documented. Thus, a value <0 indicates *incomplete documentation*. This happens more frequently when mixing `@param` and `@inheritParams`: the comments where inspected on the source and not on the compilation and, thus, the inherited parameters were not embedded. However, some functions are still missing the tag. This is not accounting for empty parameters' descriptions.

Table 8
Distribution of elements that do not document any parameters.

Sub-type	# Comments	%	Risk
Alias	1453	8.90%	Safe
Package	53	0.32%	
OOP classes	317	1.94%	
Deprecated	275	1.68%	Mid
Others	1629	9.98%	
Reexports	226	1.38%	
NULL	1771	10.85%	
NA	86	0.53%	
OOP methods	1846	11.31%	High
Functions	8660	53.07	

A complementary analysis was completed over the elements that did not have documentation related to parameters. As summarised in Table 8, about 11.16% of the elements with argument documentation are considered 'safe' since they are not intended to have them (namely, aliases, packages and classes). After that, 24.42% of elements present risk of *incomplete documentation*, since they are elements that may have arguments, such as `NA` or `NULL`, used for topic documentation, deprecated functions or methods, and reexports.

Finally, about 64.38% of the elements that do not document *any* argument are functions and methods, presenting a high risk of incompleteness. From this set, only 8.51% of the functions did not receive any argument (not even the `...`), being 'correct' in their lack of `@param` tags. However, the remainder had multiple arguments, with the top three being: two (29.86%), a single one(20.7%), and three (13.7%). There is a large number of elements without documentation, leading to *incomplete documentation*.

Findings. The analyses indicate scarce good practices. Examples are the lack of use for some anti-pattern tags, a large preference for selective import, and correct academic citations when implementing special algorithms.

However, there is a large number of practices indicating incorrect use of the Roxygen specification (e.g., lack of use of topics, tags with an incorrect purpose), outdated documentation (e.g., documenting parameters that are not part of the signature), and incomplete documentation (e.g., many empty or missing tags that should be added to a specific element).

4.4. Developers' challenges (RQ4)

This section presents the results of the survey conducted on R package developers. Its structure, preparation and distribution were discussed in Section 3.4.

Using Qualtrics, 478 emails were sent to unique email addresses; 36 emails bounced back. Though the survey obtained 72 responses, only 48 were useable (this means, it could be considered when doing the analysis), resulting in a 10.04% of response rate. To be considered as a useable response, the participant had to complete more than the demographics section of the survey, and at least the questions regarding the type of data consulted on the documentation.

4.4.1. Demographics

Regarding demographics, the participants were only asked about years of experience in R programming and the number of packages authored (regardless of CRAN availability). Responses are summarised in Table 9. As can be seen, more than half of the participants have at least five packages publicly available, and about 85.5% have 5+ years of programming experience. These results show that respondents are reasonably experienced.

Table 9
Self-reported demographics for the participants of the survey.

Question	Selection			
	<2	2–5	5–10	10+
Pkgs authored	9.09%	36.96%	18.57%	36.36%
Experience years	1.82%	12.73%	40%	45.45%

In terms of R packages, documentation can be accessed from many sources, with two being very convenient: inside the IDE (for example, by using `?functionName`), or also through online sources generated automatically with the documentation output, such as the one available on CRAN's PDF files, or rdocumentation.org. Results of this question are summarised in Fig. 7.

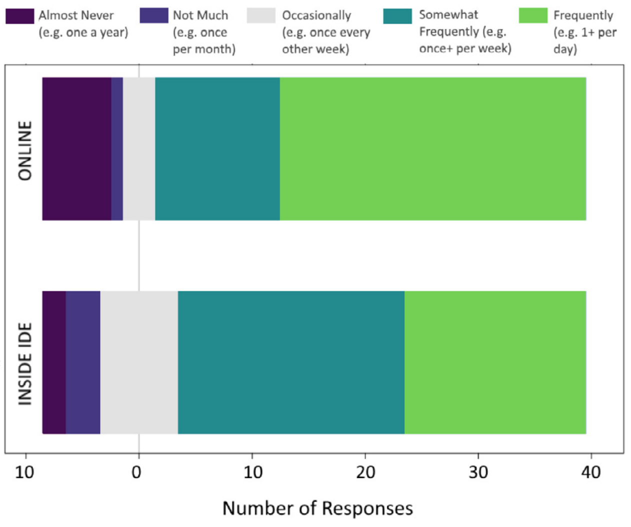


Fig. 7. Frequency of usage of documentation-generated information. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Please note that the Likert scale described in the colour legend is the same that was shown to the participants in the survey (see structure on Appendix). This structure was derived from previous works, as disclosed in Section 3.4, and thus no additional controls were applied. This may result in a threat to the validity of the

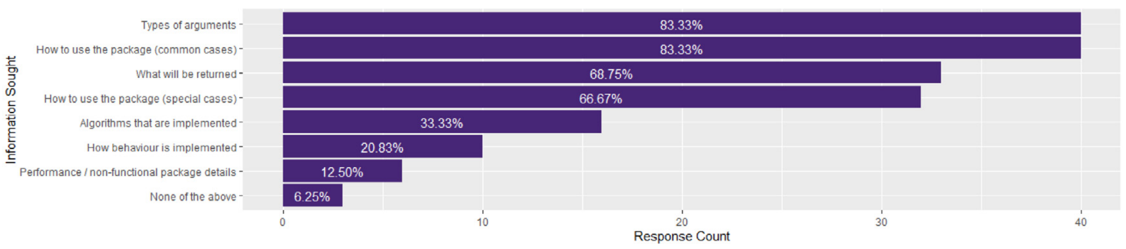


Fig. 8. Information searched when consulting the API. Percentage represents the proportion of respondents that selected the answer (multiple-choice was enabled).

response, given that it can be affected by how often the surveyed participants use R.

As seen, more than half of the participants consult the documentation inside the IDE several times a day, and almost 80% do it several times per week. Moreover, two-thirds also check externally-hosted documentation-generated sites.

When enquired about the proportion of programming time spent in consulting documentation, the reported mean was 14.5% ± 11.25%, with the outliers being 60% and 1%. This number is affected by developers' experience, and considering the demographics, it is understandable that they do not spend as much time reading documentation. This is a lower time than the reported in previous studies regarding APIs (Uddin and Robillard, 2015).

4.4.2. Content searched

Regarding meaningful information, developers tend to search for different information. The questions in this section were derived from a previous survey regarding API use (Head et al., 2018), to compare results to traditional APIs. Fig. 8 summarises these results; this was a multiple-choice question, totalling 180 selections.

For R developers, the top two options (with 83% of participants searching for each) were common use cases and types of arguments, followed by explanations of the return (almost 69%) and special use cases (almost 67%). These choices contradict traditional API documentation since studies report 'behaviour implementation' as the most searched (Head et al., 2018).

Almost 21% of participants sought behaviour. When elaborating on the specifics of what they search, unlike traditional API's reported usages (Head et al., 2018), 60% of this subgroup checks 'how arguments/parameters affect behaviour', but only 10% enquires about 'style or best practices.

This is reasonable since R programming is mostly functional programming, and (as seen in the mined data) OO capabilities are scarcely used. Nonetheless, considering the number of packages implementing specific algorithms, the lack of searches regarding how it was implemented may lead to a lack of interest in the quality of implementations. However, this is out-of-scope for this work and should be addressed in a follow-up study. As a result, the problems related to documenting parameters, examples and returns become more important given the considerable emphasis that R developers put on it.

4.4.3. Challenging documentation

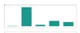
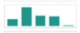
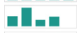
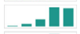
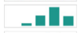
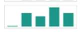
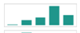
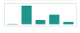
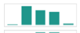
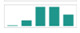
The developers were enquired about the type of documentation problems they face more often; this enquiry was done using the same structure proposed by Uddin and Robillard (2015), to compare results. Table 10 presents the results of the survey conducted in this paper. Here, the column 'frequency' presents a *sparkline*–a small embedded chart that showcases the proportion of responses for each option. The *x*-axis represents the Likert scale from 1 (left, happens 'very frequently') to 5 (right, 'extremely unlikely'). The *y*-axis is proportional to the number of maximum responses. To complement this, the column 'Mean' highlights the mean value of Likert responses.

As can be seen, the most common problems are (in order), *unexplained examples*, *ambiguity*, *incompleteness* and *fragmentation* (equally), and *tangled information*, with the most commonly reported frequency being 'somewhat likely'. In terms of documentation smells these responses support the findings about incomplete documentation (incompleteness, ambiguity and unexplained examples), and incorrect usage/writing (fragmentation and tangled).

When compared to challenges on traditional API documentation, obtained by previous studies (Uddin and Robillard, 2015), only *incompleteness*, *ambiguity* and *unexplained examples* present a barrier for developers in both studies. However, these results have some differences: R developers do not find *bloat* as a problem, but do face *fragmentation* and *tangled information* as such.

Participants were also enquired about the most common severity of the problems faced. About 56.25% said that the problems were "Moderate (kind of irritating)", and 12.5% said they were "Severe (I wasted a lot of time on this, but figured it out)". Less than 2% said they had to switch to another package. Finally, about 21% did not find these as a problem, which is aligned with the participants' reported experience.

Table 10
Challenges faced by developers when using documentation, with Likert sparklines.

	Problem	Description	Frequency	Mean*
Content	Incompleteness	The description of an element or topic wasn't where it was expected to be.		2.90
	Ambiguity	The description was mostly complete, but very unclear.		2.56
	Unexplained	A code example was insufficiently explained.		2.33
	Obsoleteness	The documentation referred to a previous version.		4.04
	Inconsistency	The documentations of elements meant to be used in combination did not agree.		3.85
	Incorrectness	Some information was incorrect.		3.40
Presentation	Bloat	The description of an API element was verbose or too extensive.		3.73
	Fragmentation	The information related to an element or topic was fragmented or scattered over too many pages or sections.		2.90
	Tangled	The description of an element or topic was tangled with information not needed.		2.96
	Redundancy	There was redundant information about the element syntax or structure.		3.60

(*) The lower the mean, the more the developers face the problem.

These results were compared to what developers find useful in the documentation (also part of the survey). It was presented as a text-entry to not bias the respondents and follow the same presentation as in other studies (Head et al., 2018; Uddin and Robillard, 2015). Like in those cases, a card-sorting approach was used to group the examples according to the elements of the documentation they were discussing. Results are summarised in Table 11.

Table 11
Helpful documentation parts, according to developers.

Topic	#
Runnable examples	11
Parameters definition	7
Goal description	7
Modularity	4
Complete descriptions	3
Correct language	2
Returns	1

The card-sorting was completed by the author and one R developer (with 5+ years of experience in R and Python). The process started with a joint scan of the responses, to discard generic cases; in total, five generic or unusable responses on the style of "all docs are useful", or "I can't think of specific examples" were removed. After that, both sorters organised the answers by documentation element (as per the Roxygen tags); this was done individually and independent of each other. When completed, both sorters had an open discussion regarding each their resulting classification. The process was completed on a single round (classification plus discussion), given that there was a single disagreement, settled during the discussion.

In terms of packages, `dplyr` and `ggplot2` where the most mentioned. As positive things to have, developers agreed that *runnable and complete examples* were the most important and useful element of documentation, followed by comprehensive parameters descriptions and general information (of the goal and intent of a package, rather than functions).

It is worth noticing that this question was optional, and not everybody answered it. The following are some of the comments received in this question (sic):

- "Examples, examples, examples! Code examples that show all the potential variations in how a function can be used with the various arguments set at each possible value are the most helpful thing".
- "Closely related functions documented on the same page; other relevant functions presented in a 'See Also' section. Runnable examples illustrating functionality without being too arcane".
- "[...] written in a plain language, have little jargon used, they are concise, and they are designed well (from a technical UX perspective and legibility)". The acronym UX is assumed to mean 'user experience'.
- "I find projects that offer complete, copy-pastable examples really valuable... much more valuable than examples that don't show all library imports, have fake file paths like `/path/to/file`, and other things you have to figure out".
- "Arguments are described with type and function. The name of the argument matches the function. Examples are not just a list of possibilities, but show the usability and explain use cases".

Findings. The survey participants are experts, based on reported experience. About half consult documentation several times per day, with 14.1% of their coding time dedicated to reading it.

Unlike traditional APIs, the most searched content is the type of arguments and common use cases (about 83% of respondents on each). By combining responses, searches for parameters, returns and examples are the most common.

The major challenges with documentations are unexplained examples, ambiguity and incompleteness. Unlike with APIs, bloatedness is not an issue.

5. Discussion

There are other ways to generate documentation for an R package (such as Latex-style or directly writing the Rmd files). However, Roxygen was chosen to narrow the scope because it is widely recommended in popular R books (Wickham and Grolemund, 2017; Peng, 2012), and its similarities to Javadoc allowed a comparison to previous results.

Furthermore, this study did not cover all of the available sections, purposefully excluding the 'title' and 'description' section. This decision was based on several reasons. First, these elements are not always tagged (the tags are optional) and can span several paragraphs without additional tags, requiring more complex detection. Second, only extracting tagged cases could bias the results. Third, these are (by design) large text-based elements that warrant a detailed analysis, which would have extended this study considerably. As a result, further analysis on these sections (namely, `@title`, `@description` and `@section`, regardless of the presence of the tag) are a future work.

To organise the main contribution of this paper, Table 12 presents the findings according to the smells that were investigated to answer the research questions. It is worth mentioning that, though several of the enquiries and analysis seem to indicate the presence of *outdated documentation* smell, this is out-of-scope and considered a future-work. Likewise, albeit not a smell, several cases of *incorrect use* (of Roxygen's specification) were detected and thus included.

The findings indicate that R package documentation tends to be incomplete, with many cases of incorrect use of the Roxygen tags; in some cases, like the `@author` tag, it can be attributed to poor documentation of the tool. Moreover, several problems were identified with regards to the examples, parameters and return– which are identified by the survey partakers responded that these are the most valuable parts of the documentation and the most queried as well. Though participants did not consider the existing problems as dangerous blockers, they also reported being very experienced, which may have affected this perception.

Moreover, participants' use and perception of the documentation vary considerably when compared to traditional API documentation (Head et al., 2018; Uddin and Robillard, 2015).

A key point is that R is package-based language where developers are forced to download packages to be capable of writing advanced code. However, there is a concerning amount of Documentation Debt in the documentation offered by developers in existing packages; this can lead to misuse and misunderstanding of functionalities, thus transitioning to low quality code. Prior work determined that Documentation Debt is the most commonly addressed TD type when peer-reviewing R packages (Codabux et al., 2021); that study assessed only rOpenSci, but the organisation's relevance among the R community is known for setting *de facto* standards among this public. Outside of R programming, other authors have demonstrated that documentation

Table 12
Types of documentation debt smells found in the analysis, and their reasons.

Smell	Reason
Non-existent (RQ2)	Near 9% of the R packages have no documentation (using any API format), and less than 1% continue to use the original latex-style. About 17.6% of exported elements have no documentation.
Incomplete (RQ3)	Many of the tags are part of the documentation but are incomplete, having an empty description. This is the case of `@seealso` (12%), `@reference` (33.2%), and `@returns` (7.8%). There are cases of packages not using a specific tag in *any* of its documented elements. For instance, examples (missing on 6.4% of the packages), `@return` (in 9% of packages), parameters (in any form, about 6% of the packages). 76.5% of the elements that do not document returns are methods or functions that should have this information. Likewise, close to 64.4% of the elements that do not document parameters are functions or methods, and only 8.5% of those have zero arguments. Most functions that document parameters are missing arguments or document non-existent arguments (see Fig. 6).
Incorrect[a]	Almost 5% of the `@seealso` tags are either missing the link or used to refer to academic citations, disregarding the use of this tag. Moreover, this tag is scarcely combined with `@family`, missing the purpose of simplifying the documentation. About 17.4% of the instances of `@example` are misused, embedding the code rather than linking it to a file. There is limited use of Roxygen importing (with either tag), which may lead to function cloning a manual edition of the `namespace` file. About 200 comments with the tag `@return` had two or more instances of the tag in the same element. The `@author` tag is available in Roxygen, recommended in several documents, but not listed in the official Roxygen documentation. This is a problem of incomplete specification of the API that can hinder/mislead its use.
Outdated[b]	The tag `@s3method` is deprecated, indicating that the documentation may be outdated. Almost 39% of all examples (with `@example`) have a `\dontrun`, which prevents CRAN checks from running the example. It may lead to outdated or incorrect examples.

[a]This is not a formally acknowledged smell but was a problem found in the analysis.
[b]It is a smell not formally explored in this paper (out of scope), but some factors may indicate its presence.

quality is one of the factors leading to better overall software quality (Gorla and Lin, 2010; Chomal and Saini, 2014).

Moreover, while R is commonly used for scientific software, 'how a behaviour is implemented' is scarcely sought, according to the survey. This may pose an issue, as developers rely on what is said to be coded instead of verifying it through the documentation (e.g., examples, descriptions). As a result, this also opens another path for future works.

As a result, several key *implications* can be summarised from this work.

- For **practitioners**, this paper is a call to action to detect the weakness in the documentation they produce, and improve it. Given that documentation is a key factor for R package review (Codabux et al., 2021), dedicated organisations (e.g., rOpenSci, BioConductor, and even CRAN) can leverage the findings of this paper (e.g., Table 12) as part of their review process. Likewise, companies and research groups producing R packages can use the results of this study as a checklist to assess the quality of the documentation they produce. Finally, given Roxygen is an open-source package, willing contributors (i.e., R developers) could use these findings to add new functionality to the package `roxygen2`, distribute changes and update its documentation. Given the extent of R developers' groups (e.g., RLadies, RUGs, among others), unifying and consolidating advice regarding documentation through their meetings (e.g., between books and Roxygen itself) will also assist junior developers.
- For **researchers**, this paper opens multiple paths of future works. As mentioned before, further analysis on titles and descriptions of Roxygen documentations is needed, alongside reasons on why behaviour implementation is seldom searched for. Additionally, this work can be extended to foster better documentation practices in basic programming courses for computational science, and analyse its impact to improve the next generation of R users. Further efforts should be devoted to survey R developers (and scientific software developers) to understand their documentation needs, challenges and weaknesses; likewise, understanding the documentation process in scientific software, and its different to traditional (non-scientific) software is essential. Along those lines, the findings of this paper should be extended to other scientific programming languages, to be considered during the assessment of software artifacts during the peer-review process. Likewise, working with data-science consultancy industries to translate this line of research will assist in the development of industry-ready processes to improve documentation practices, that are also time and cost effective.

5.1. Threats to validity

The following threats were considered in this work.

External Validity. These relate to the generalisability of the results. The dataset consisted of 379 systematically-selected, open-source R packages mined from GitHub. Still, it is unclear if the findings would generalise to all R packages. Furthermore, the survey respondents sample may not represent the entire population of developers, and thus results might not generalise to all of them. This was minimised by selecting a statistically relevant sample of packages (with 95% confidence and 5% error) and establishing a clear inclusion and exclusion criteria. To the best of the author's knowledge, this is the first and largest MSR regarding Roxygen documentation practices in R packages.

Internal Validity. These concern the conditions under which experiments are performed. Even though Roxygen has several similarities to Javadoc, there was no baseline data to use as training sets for automated algorithms specific to R programming. Though many automated studies were conducted on the whole set, they may not be entirely generalisable.

Though the use of GitHub's 'best match' sorting approach is a common standard, there is no clearly defined algorithm for it. Thus, as with any other search in GitHub, its use is a threat to validity of the reproducibility and generalisability of the data collection and sampling approach, given that the order obtained for this paper may not be reproducible on another date. However, this study was subject to an Ethical Protocol, and the survey participants' were offered anonymity; thus, further analysis on how the search may have varied over time, were not conducted (Gan et al., 2018). However, it is possible to assume that: (a) they may have more commits, stars, forks or issues (which could affect the search order); (b) that the documentation may have changed as the software evolved (an expected aspect of documentation (Tracz, 2015)); or (c) that they may remain the same (given the difference on maintenance of scientific software compared to more traditional software (Hannay et al., 2009)). In any case, the risk of considerable changes in the short-term that could lead to insignificant results, is negligible.

Likewise, as the survey structure was replicated from a previous work (see Section 3.4), the Likert scales remained unchanged. As a result, the responses to Question 1 (see Appendix) may have the developers' personal bias, as the answer may change depending on how often they use R.

Construct validity. These concern the relationship between theory and observation. A threat of using package documentation is the consistency of changes and the use of natural language itself. Therefore, results may be impacted by the quantity and quality of comments in every R package. Besides, since this study parted from the definitions of *documentation debt* created for OOP, it is possible that 'data science oriented' types of TD were not evaluated. However, this study was aimed to be a first exploratory approach, leaving room for future works.

6. Conclusion

R is a package-based programming ecosystem primarily targeted to statistics and data science that provides a streamlined way to reuse third-party code and extend its functionalities. Therefore, R developers are highly dependent on the API documentation provided alongside those packages, as it allows them to use the new functionalities correctly. *Documentation Technical Debt* (DTD) has been identified as shortcuts (non-optimal

Table 13
Full survey structure (questions, and answers types).

Question	Style	Possible answers
Demographics		
A. How many R packages have you authored?	Single choice	<2, 2–5, 5–10, >10
B. How many years of experience do you have as an R programmer?	Single choice	<2 years, 2–5 years, 5–10 years, 10+ years
Documentation		
1. How often do you...? (1.1) Consult existing documentation from inside RStudio, (1.2) Consult online references created with documentation	5-Likert Matrix	Almost Never (e.g., once a year) Not Much (e.g., once per month) Occasionally (e.g., once every other week) Somewhat Frequently (e.g., once per week) Frequently (e.g., 1+ per day)
2. What best describes the information you are look for?	Multi-choice (≤ 3)	Package usage (common cases) Package usage (special cases) Types of arguments Performance/non-functional details How a behaviour is implemented (*) Algorithms that are implemented What will be returned None of the above [comment]
3. If you look at behaviours/algorithms. Why are looking into it? [if (*) was selected]	Multi-choice (≤ 3)	To understand unexpected code behaviour Finding code or logic to reuse Planning a code refactoring Checking style or best practices Checking how arguments affect behaviour Other [comment]
4. What proportion of your programming time is spent consulting documentation?	Bar	[Input %]
5. Which of the following problems have you faced when consulting package documentation? (5.1) An element/topic description was not where I expected it. (5.2) The description was mostly complete, but very unclear. (5.3) A code example was insufficiently explained. (5.4) The documentation referred to a previous version. (5.5) Documentations of combinable elements did not agree. (5.6) Some information was incorrect. (5.7) An API element description was verbose/too extensive. (5.8) The information related to an element or topic was fragmented or scattered over too many pages or sections. (5.9) The description of an element or topic was tangled with information you did not need. (5.10) Redundant information about elements' syntax or structure	5-Likert Matrix	Never–Always
6. How severe was the documentation problem to complete your development task, when you last observed it?	Single choice	Not a problem Moderate (kind of irritating) Severe (I wasted a lot of time on this, but figured it out) Blocker (I could not get past and picked another package) No opinion Other [comment]
7. Give up to three examples of documentation that you find useful. Explain why the documentation was useful.	Textbox	[text-box]
8. Any comment about your experience with documentation in R packages?	Textbox	[text-box]

decisions) related to documentation and can be non-existent, incomplete or outdated (namely, smells).

To the author's knowledge, this is the first study that reports on the current state of documentation of 379 systematically-selected, open-source R packages available in GitHub. During this process, this research also surveyed the developers of these packages to understand the documentation culture and determine the challenges they face while coding.

The findings indicate that *incomplete documentation* is a persistent smell also identified by the developers, presenting problems related to incorrect use of the Roxygen specification, unexplained examples and fragmented information. Moreover, R developers' priorities in documentation sections are reportedly different from those of traditional API users. The low quality of documentation poses a challenge for developers, considering R's intrinsic, quasi-mandatory use of packages while programming.

Several lines of future works can be addressed. Some documentation sections were not explored (namely, title, description and sections) and should be further evaluated. Albeit the findings indicate cases of *outdated documentation*, this was not actively explored and should be investigated by comparing the evolution of the documents over time through version control commits. Finally, regardless of R mainly being used for scientific software, developers appear unconcerned about how behaviour and algorithms are implemented; future studies should aim to understand the reason and consequences of this.

CRediT authorship contribution statement

Melina Vidoni: Conceptualisation, Methodology, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualisation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This research did not receive any specific grants from funding agencies in the public, commercial, or not-for-profit sectors. The author thanks the anonymous reviewers who provided helpful comments on earlier drafts of the manuscript.

Ethical considerations

The methodology used in this manuscript, described in the sections below, was revised approved by the Australian National University's Human Ethics Research Committee (HREC), with project code 2021–24181.

Appendix. Survey structure

Table 13 presents the questions, types of answers and available answers for the developers' survey introduced in Section 3.4.

References

Alves, N.S.R., Ribeiro, L.F., Caires, V., Mendes, T.S., Spínola, R.O., 2014. Towards an ontology of terms on technical debt. In: 2014 Sixth International Workshop on Managing Technical Debt. pp. 1–7. http://dx.doi.org/10.1109/MTD.2014.9.

Blasi, A., Kuznetsov, K., Goffi, A., Castellanos, S.D., Gorla, A., Ernst, M.D., Pezzè, M., 2017. Semantic-based analysis of javadoc comments. In: Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017. SATToSE, Madrid, Spain, pp. 1–5.

Borges, H., Valente, M.T., 2018. What's in a GitHub star? Understanding repository starring practices in a social coding platform. J. Syst. Softw. 146, 112–129. http://dx.doi.org/10.1016/j.jss.2018.09.016.

Chomal, V.S., Saini, J.R., 2014. Significance of software documentation in software development process. Int. J. Eng. Innov. Res. 3 (4), 410.

Claes, M., Mens, T., Tabout, N., Grosjean, P., 2015. An empirical study of identical function clones in CRAN. In: 2015 IEEE 9th International Workshop on Software Clones (IWSC). pp. 19–25. http://dx.doi.org/10.1109/IWSC.2015.7069885.

Codabux, Z., Vidoni, M., Fard, F., 2021. Technical debt in the peer-review documentation of r packages: a rOpenSci case study. In: 2021 International Conference on Mining Software Repositories. IEEE, Madrid, Spain, pp. 1–11.

Decan, A., Mens, T., Claes, M., 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 2–12. http://dx.doi.org/10.1109/SANER.2017.7884604.

Gan, W., Chun-Wei, J., Chao, H.-C., Wang, S.-L., Yu, P.S., 2018. Privacy preserving utility mining: A survey. In: 2018 IEEE International Conference on Big Data (Big Data). pp. 2617–2626. http://dx.doi.org/10.1109/BigData.2018.8622405.

German, D.M., Adams, B., Hassan, A.E., 2013. The evolution of the R software ecosystem. In: 2013 17th European Conference on Software Maintenance and Reengineering. pp. 243–252. http://dx.doi.org/10.1109/CSMR.2013.33, iSSN: 1534-5351.

Gorla, N., Lin, S.-C., 2010. Determinants of software quality: A survey of information systems project managers. Inf. Softw. Technol. 52 (6), 602–610. http://dx.doi.org/10.1016/j.infsof.2009.11.012, URL: https://www.sciencedirect.com/science/article/pii/S0950584909002122.

Hannay, J.E., MacLeod, C., Singer, J., Langtangen, H.P., Pfahl, D., Wilson, G., 2009. 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. http://dx.doi.org/10.1109/SECSE.2009.5069155.

Head, A., Sadowski, C., Murphy-Hill, E., Knight, A., 2018. When not to comment: Questions and tradeoffs with API documentation for C++ projects. In: Proceedings of the 40th International Conference on Software Engineering. In: ICSE '18, Association for Computing Machinery, New York, NY, USA, pp. 643–653. http://dx.doi.org/10.1145/3180155.3180176.

Hornik, K., 2012. Are there too many R packages? Aust. J. Statist. 41 (1), 59–66.

Jiang, J., Lo, D., He, J., Xia, X., Kochhar, P.S., Zhang, L., 2017. Why and how developers fork what from whom in GitHub. Empir. Softw. Eng. 22 (1), 547–578. http://dx.doi.org/10.1007/s10664-016-9436-6.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D., 2014. The promises and perils of mining GitHub. In: Proceedings of the 11th Working Conference on Mining Software Repositories. In: MSR 2014, Association for Computing Machinery, New York, NY, USA, pp. 92–101. http://dx.doi.org/10.1145/2597073.2597074.

Korkmaz, G., Kelling, C., Robbins, C., Keller, S.A., 2018. Modeling the impact of R packages using dependency and contributor networks. In: 2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM). pp. 511–514. http://dx.doi.org/10.1109/ASONAM.2018.8508255, iSSN: 2473-991X.

Korkmaz, G., Kelling, C., Robbins, C., Keller, S., 2019. Modeling the impact of Python and r packages using dependency and contributor networks. Soc. Netw. Anal. Min. 10 (1), 7. http://dx.doi.org/10.1007/s13278-019-0619-1.

Li, K., Chen, P.-Y., Yan, E., 2019. Challenges of measuring software impact through citations: An examination of the lme4 R package. J. Informetr. 13 (1), 449–461. http://dx.doi.org/10.1016/j.joi.2019.02.007.

Li, K., Yan, E., 2018. Co-mention network of R packages: Scientific impact and clustering structure. J. Informetr. 12 (1), 87–100. http://dx.doi.org/10.1016/j.joi.2017.12.001.

Lu, Y., Mao, X., Li, Z., Zhang, Y., Wang, T., Yin, G., 2018. Internal quality assurance for external contributions in GitHub: An empirical investigation. J. Softw.: Evol. Process. 30 (4), e1918. http://dx.doi.org/10.1002/smr.1918, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1918 URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1918 e1918 smr.1918.

Maalej, W., Robillard, M.P., 2013. Patterns of knowledge in API reference documentation. IEEE Trans. Softw. Eng. 39 (9), 1264–1282. http://dx.doi.org/10.1109/TSE.2013.12.

Medappa, P., Srivastava, S., 2018. Restrictions in open source: A study of team composition and ownership in open source software development projects. In: Proceedings of the 39th International Conference on Information Systems. Association for Information Systems, United States, pp. 1–12.

Peng, R., 2012. R Programming for Data Science. Lulu.com, URL: https://books.google.com.au/books?id=GSePDAEACAAJ.

Plakidas, K., Schall, D., Zdun, U., 2017. Evolution of the r software ecosystem: Metrics, relationships, and their impact on qualities. J. Syst. Softw. 132, 119–146. http://dx.doi.org/10.1016/j.jss.2017.06.095.

Rios, N., Mendes, L., Cerdeiral, C., Magalhães, A.P.F., Perez, B., Correal, D., Astudillo, H., Seaman, C., Izurieta, C., Santos, G., Oliveira Spínola, R., 2020. Hearing the voice of software practitioners on causes, effects, and practices to deal with documentation debt. In: Madhavji, N., Pasquale, L., Ferrari, A., Gnesi, S. (Eds.), Requirements Engineering: Foundation for Software Quality. Springer International Publishing, Cham, pp. 55–70.

Singer, J., Sim, S.E., Lethbridge, T.C., 2008. Software engineering data collection for field studies. In: Guide to Advanced Empirical Software Engineering. Springer London, London, pp. 9–34. http://dx.doi.org/10.1007/978-1-84800-044-5_1.

Stulova, N., Blasi, A., Gorla, A., Nierstrasz, O., 2020. Towards detecting inconsistent comments in java source code automatically. In: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, Adelaide, Australia, pp. 65–69. http://dx.doi.org/10.1109/SCAM51674.2020.00012.

Tan, L., 2015. Chapter 17 - code comment analysis for improving software quality. In: Bird, C., Menzies, T., Zimmermann, T. (Eds.), The Art and Science of Analyzing Software Data. Morgan Kaufmann, Boston, pp. 493–517. http://dx.doi.org/10.1016/B978-0-12-411519-4.00017-3.

Tan, S.H., Marinov, D., Tan, L., Leavens, G.T., 2012. @Tcomment: Testing javadoc comments to detect comment-code inconsistencies. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE, Montreal, Canada, pp. 260–269. http://dx.doi.org/10.1109/ICST.2012.106.

Tracz, W., 2015. Refactoring for software design smells: Managing technical debt by girish suryanarayana, ganesh samarthyam, and tushar sharma. SIGSOFT Softw. Eng. Notes 40 (6), 36. http://dx.doi.org/10.1145/2830719.2830739.

Turcotte, A., Vitek, J., 2019. Towards a type system for R. In: Proceedings of the 14th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems. In: ICOOOLPS '19, Association for Computing Machinery, London, United Kingdom, pp. 1–5. http://dx.doi.org/10.1145/3340670.3342426.

Uddin, G., Robillard, M.P., 2015. How API documentation fails. IEEE Softw. 32 (4), 68–75. http://dx.doi.org/10.1109/MS.2014.80.

Vidoni, M., 2021a. Evaluating unit testing practices in r packages. In: Proceedings of the 43rd International Conference on Software Engineering (ICSE). IEEE, Madrid, Spain, pp. 1–12.

Vidoni, M., 2021b. Self-admitted technical debt in R packages: An exploratory study. In: 2021 International Conference on Mining Software Repositories. IEEE, Madrid, Spain, pp. 1–11.

Wickham, H., Grolemund, G., 2017. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data, 1st O'Reilly Media, Inc., USA.

Zanella, G., Liu, C.Z., 2020. A social network perspective on the success of open source software: The case of r packages. In: Hawaii International Conference on System Sciences. pp. 471–480. http://dx.doi.org/10.24251/HICSS.2020.058.

Dr **Vidoni** is a newly joined academic (Lecturer, eq. to Assistant Professor) at the Australian National University in the School of Computing, where she continues her domestic and international collaborations with Canada and Germany. Dr Vidoni's main research interests are mining software repositories, technical debt and software development; empirical software engineering when applied to data science and scientific software.

She graduated from Universidad Tecnologica Nacional (UTN) as an Information Systems Engineer. Years later, she received her Ph.D. on the same institution, with the maximum qualification. She funded R-Ladies Santa Fe in 2018, and in 2019 moved to Australia to further her research career. Since 2018, she is also Associate Editor for rOpenSci, often acting as Editor in Chief.