

OPEN
ANALYSIS

A large-scale study on research code quality and execution

Ana Trisovic¹✉, Matthew K. Lau², Thomas Pasquier³ & Mercè Crosas¹

This article presents a study on the quality and execution of research code from publicly-available replication datasets at the Harvard Dataverse repository. Research code is typically created by a group of scientists and published together with academic papers to facilitate research transparency and reproducibility. For this study, we define ten questions to address aspects impacting research reproducibility and reuse. First, we retrieve and analyze more than 2000 replication datasets with over 9000 unique R files published from 2010 to 2020. Second, we execute the code in a clean runtime environment to assess its ease of reuse. Common coding errors were identified, and some of them were solved with automatic code cleaning to aid code execution. We find that 74% of R files failed to complete without error in the initial execution, while 56% failed when code cleaning was applied, showing that many errors can be prevented with good coding practices. We also analyze the replication datasets from journals' collections and discuss the impact of the journal policy strictness on the code re-execution rate. Finally, based on our results, we propose a set of recommendations for code dissemination aimed at researchers, journals, and repositories.

Introduction

Researchers increasingly publish their data and code to enable scientific transparency, reproducibility, reuse, or compliance with funding bodies, journals, and academic institutions¹. Reusing data and code should propel new research and save researchers' time, but in practice, it is often easier to write new code than reuse old. Even attempting to reproduce previously published results using the same input data, computational steps, methods, and code has shown to be troublesome. Studies have reported a lack of research reproducibility^{2,3} often caused by inadequate documentation, errors in the code, or missing files.

Paradigms such as literate programming could help in making the shared research code more understandable, reusable, and reproducible. In literate programming, traditional source code is interspersed with explanations of its logic in a natural language⁴. The paradigm was encouraged for scientific computing and data science to facilitate reproducibility and transparency. However, in practice, researchers write code intending to obtain scientific insights, and there is often no incentive to structure and annotate it for reuse. As a result, the research code quickly becomes unusable or unintelligible after meeting its initial purpose⁵.

Though much of the code's intrinsic design will determine its longevity, its dissemination platform could also have a compelling influence⁶. In particular, data and code repositories are some of the primary venues for sharing research materials. They aim to support researchers by creating general dissemination guidelines and descriptive metadata, but they cannot always prevent irreproducibility and code-rot due to the vast diversity of programming languages and complex computing processes. This is only aggravated as researchers generate and share new results and code at a rate higher than ever before.

This paper presents a study that provides an insight into the programming literacy and reproducibility aspects of shared research code. The first premise of the study is to examine the properties of the shared datasets and research code. Information such as their size, content, presence of comments in the code, and documentation in the directory help us understand the current state of research code. By comparing the observed coding practices to the established best practices, we identify the existing weak points and areas of improvement for researchers writing code. Our content analysis gives us an insight into the storage needs and requirements for supporting files, such as documentation, images, or maps. The second premise of the study is to examine what happens when an external researcher retrieves and re-executes shared research code. In particular, we ask what

¹Institute for Quantitative Social Science, Harvard University, Cambridge, MA, USA. ²CAS Key Laboratory of Forest Ecology and Management, Institute of Applied Ecology, Chinese Academy of Sciences, Shenyang, China.

³Department of Computer Science, University of British Columbia, Vancouver, BC, Canada. ✉e-mail: anatrivic@harvard.edu

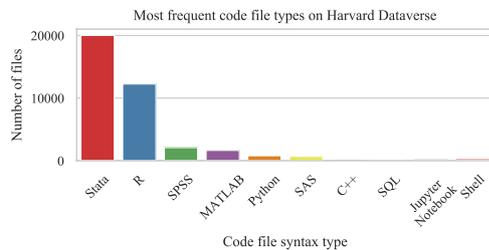


Fig. 1 Most popular code file types on Harvard Dataverse (Oct, 2020). Of the top two, R is open source and free.

the common errors are in executing this code and whether they can be solved with simple changes in the code. We explore if the code re-execution rates vary between different disciplines and other available features, and analyze the practices behind the best-performing ones. Finally, we explore code re-execution as a required but not sufficient condition for reproducibility. Based on the study's findings, we conclude with recommendations for disseminating research code for researchers, journals, and repositories.

Background

Our study uses code deposited and shared at the Harvard Dataverse repository. The Dataverse project (<http://dataverse.org>) is an open-source data repository platform for sharing, archiving, and citing research data. It is developed and maintained by the Harvard's Institute for Quantitative Social Sciences (IQSS) and a community of open source contributors. Currently, more than 60 institutions worldwide run Dataverse instances as their data repository, each hosting data generated by one or more institutions.

Dataverse repositories allow researchers to deposit and share all research objects, including data, code, documentation, or any combination of these files. A bundle of these files associated with a published scientific result is called a replication package (or "replication data" or dataset in Dataverse repositories). Researchers' code from replication packages usually operates on data to obtain the published result. For the Harvard Dataverse repository, replication packages are typically prepared and deposited by researchers themselves in an unmediated fashion (self-curated).

The most popular programming languages among the Harvard Dataverse repository users are Stata and R, as shown from the frequency of deposited code files in Fig. 1. The two languages are often used in quantitative social science research. Their observed popularity can be attributed to Harvard Dataverse repository initially specializing in sharing social science research data. In the last five years, it has become a general-purpose, inter-disciplinary data repository. Stata is proprietary statistical software used in economics, sociology, political science, and health sciences. R is free and open-source software frequently used among statisticians and data analysts in the social sciences. Due to its popularity among academics and its open-source license, R is an ideal candidate for our study. It is currently ranked as the 13th most popular language in the TIOBE index (<https://www.tiobe.com/tiobe-index>). In the past, it was ranked as the most popular language⁷ and has been rated among the top in the Kaggle Machine Learning & Data Science Survey in the previous few years⁸. R originated as an open-source and free version of S, a statistical command language that made programming accessible without the necessity of formal training. R is highly adaptable due to its extensible package system, which led to a surge of community-driven developments. Although the broad community development created potential for unsustainable code, methods for package standardization and quality control have been improving with the creation of RStudio, an integrated development environment (IDE) for R, and online communities like R-Hub and ROpenSci.

Implementation and Methods

The R programming language is the main focus of our study due to its open-source license and popularity in scientific computing. We retrieve the content of 2109 publicly-available replication packages published from 2010 to July 2020 that contain 9078 R code files from the Harvard Dataverse repository. The Harvard Dataverse archives more than 40,000 datasets containing over 500,000 files at the time of writing. The rest of the datasets, over 65,000, are harvested from other federated repositories. For our analysis, we use only the deposited datasets (not harvested) due to the metadata differences across different repositories. Below, we elaborate on the study's implementation, workflow, and data collection.

We use AWS Batch (<https://aws.amazon.com/batch>) to parallelize the effort of retrieving and re-executing research code in each replication package. AWS Batch automatically provisions resources and optimizes the workload distribution while executing jobs without interactions with the end-user. All replication packages in the Harvard Dataverse repository are uniquely identified with a DOI (digital object identifier), and we start the analysis by retrieving the list of DOIs that contain R code (Fig. 2).

1. The DOI list is used to define the AWS jobs, which are then sent to the batch queue, waiting until resources become available for their execution.
2. When a job leaves the queue, it instantiates a pre-installed Docker image that contains the necessary software pipeline to retrieve a replication package and execute its R code.
3. Each job re-executes code from a single replication package using an Amazon EC2 instance with 16 vCPUs and 1024 GB of memory.

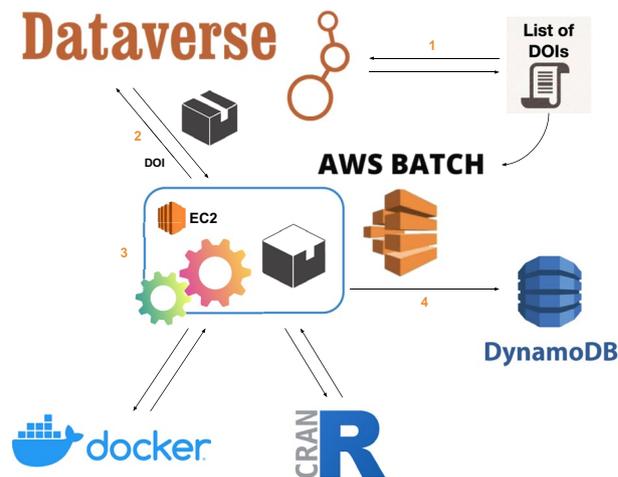


Fig. 2 Implementation on the AWS Batch.

4. Finally, the results and information related to the re-execution are stored on DynamoDB (<https://aws.amazon.com/dynamodb>).

The collected data, source code and complete instructions to reproduce our analysis are available online at Dataverse⁹ and GitHub under MIT license.

Data collection workflow. For testing research code re-execution, we use a Docker image with pre-installed conda environment manager, R and Python software on Debian GNU/Linux 10. The image contains three independent R environments, each with a different version of the R interpreter and corresponding r-essentials, a bundle of approximately 200 most popular R packages for data science. In addition to the software, the image contains a custom-made workflow that conducts the study and collects data. The logic of the workflow is the following:

1. It downloads a replication package from the Harvard Dataverse repository. We verify and note if the file has correctly downloaded or if there was a checksum error. We collect data on the size and content of the replication package.
2. We conduct an automatic code cleaning, scanning and correcting the code for some of the most common execution errors, such as hard-coded path variables (see the next section). Statistics on code files, such as the number of lines, libraries, and comments, are also collected.
3. The workflow attempts to execute the researchers' code for an allocated period of one hour per file and five hours in total. The re-execution test is conducted with and without the code cleaning step, and the result (success, error, or time-limit exceeded) is recorded.
4. The re-execution results and other collected data are passed to the backend database for analysis.

Though a total of 2,170 replication packages contained R code and were visible through the Dataverse API, we successfully retrieved 2109 (97%) of them. Some of these packages had restricted access and caused an 'authorization error' when we attempted to retrieve them. In other cases, files had obscure and erroneous encoding, which caused errors during the download. Those were excluded from our study.

Code cleaning. Our implementation of code cleaning aims to solve some of the most common re-execution errors. In particular, it removes absolute file paths, standardizes file encoding, and identifies and imports used libraries to set up a proper execution environment. The research code is modified to install the used library if it is not already present in the environment. The code cleaning approach is kept relatively simple to minimize the chance of 'breaking the code' or creating errors that were not previously there. Readers can learn more about the technical implementation of code cleaning in Appendix 3.

Results and Discussion

We define ten research questions to provide a framework for the study. The first group of questions revolves around coding practices (RQ 1–3), while the other around the automated code re-execution (RQ 4–10).

RQ 1. What are the basic properties of a replication package in terms of its size and content? Our first research question focuses on the basic dataset properties, such as its size and content. The average size of a dataset is 92 MB (with a median of 3.2 MB), while the average number of files in a dataset is 17 files (the median is 8). Even though it may seem that there is a large variety between datasets, by looking at the distributions, we observe that most of the datasets amount to less than 10 MB (Fig. 3a) and contain less than 15 files (Fig. 3b).

Analyzing the content of replication packages, we find that about 40% of them (669 out of 2,091) contain code in other programming languages (i.e., not R). Out of 2091 datasets, 620 contained Stata code (.do files), 46

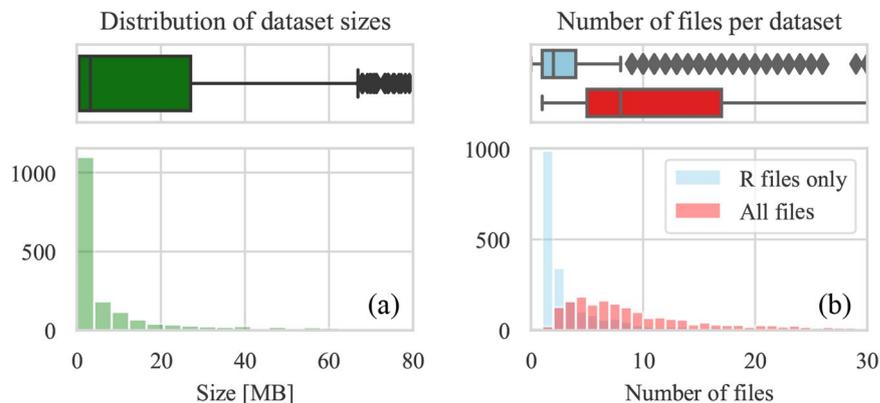


Fig. 3 Dataset sizes and file counts.

had Python code (.py files), and 9, 7, and 6 had SAS, C++, and MATLAB code files, respectively. The presence of different programming languages can be interpreted in multiple ways. It might be that a dataset resulted in a collaboration of members who preferred different languages, i.e., one used R and another Stata. Alternatively, it may be that different analysis steps are seamlessly done in different languages, for example, data wrangling in R and visualization in Python. However, using multiple programming languages may hinder reproducibility, as an external user would need to obtain all necessary software to re-execute the analysis. Therefore, in the re-execution stage of our study, it is reasonable to expect that replication packages with only R code would perform better than those with multiple programming languages (where R code might depend on the successful execution of the code in other languages).

The use of R Markdown and Rnw have been encouraged to facilitate result communication and transparency¹⁰. R Markdown (Rmd) files combine formatted plain text and R code that provide a narration of research results and facilitate their reproducibility. Ideally, a single command can execute the code in an R Markdown file to reproduce reported results. Similarly, Rnw (or Sweave) files combine content such as R code, outputs, and graphics within a document. We observe that only a small fraction of datasets contain R Markdown (3.11%) and Rnw files (0.24%), meaning that to date few researchers have employed these methods.

Last, we observe that 91% of the files are encoded in ASCII and about 5% in UTF-8. The rest use other encodings, with ISO-885901 and Windows-1252 being the most popular (about 3.5% together). In the code cleaning step, all non-ASCII code files (692 out of 8173) were converted into ASCII to reduce the chance of encoding error. In principle, less popular encoding formats are known to sometimes cause problems, so using ASCII and UTF-8 encoding is often advised¹¹.

RQ 2. Does the research code comply with literate programming and software best practices? There is a surge of literature on best coding practices and literate programming^{12–16} meant to help developers create quality code. One can achieve higher productivity, easier code reuse, and extensibility by following the guidelines, which are typically general and language agnostic. In this research question, we aim to assess the use of best practices and programming literacy in the following three aspects: meaningful file and variable naming, presence of comments and documentation, and code modularity through the use of functions and classes.

Best practices include creating descriptive file names and documentation. Indeed, if file names are long and descriptive, it is more likely that they will be understandable to an external researcher. The same goes for additional documentation within the dataset. We observe that the average filename length is 17 characters without the file extension, while the median is 16. The filename length distribution (Fig. 4a) shows that most file lengths are between 10 and 20 characters. However, we note that about a third or 32% (669) of file names contain a ‘space’ character, which is discouraged as it may hinder its manipulation when working from the command line. We also searched for a documentation file, or a file that contains “readme”, “codebook”, “documentation”, “guide” and “instruction” in its name, and found it in 57% of the datasets (Fig. 4b). The authors may have also adopted a different convention to name their documentation material. Therefore, we can conclude that the majority of authors upload some form of documentation alongside their code.

Further, we examine the code of the 8875 R files included in this study. The average number of code lines per file is 312 (the median is 160) (Fig. 5a). Considering that there are typically 2 R files per dataset (median, mean is 4), we can approximate that behind each published dataset lies about 320 R code lines.

Comments are a frequent part of code that can document its processes or provide other useful information. However, sometimes they can be redundant or even misleading to a reuser. It is good practice to minimize the number of comments not to clutter the code and replace them with intuitive names for functions and variables¹². To learn about commenting practices in the research code, we measure the ratio of code lines and comment lines for each R file. The median value of 4.5 (average is 7) can be interpreted as one line of comments documenting 4 or 5 lines of code (Fig. 5b). In other words, we observe that comments comprise about 20% of the shared code. Though the optimal amount of comments depends on the use case, a reasonable amount is about 10%, meaning that the code from Harvard Dataverse is on average commented twice as much.

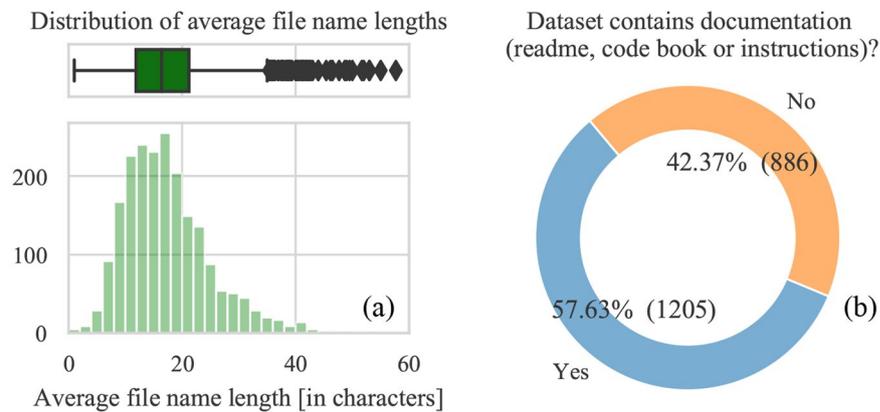


Fig. 4 File name lengths and presence of documentation.

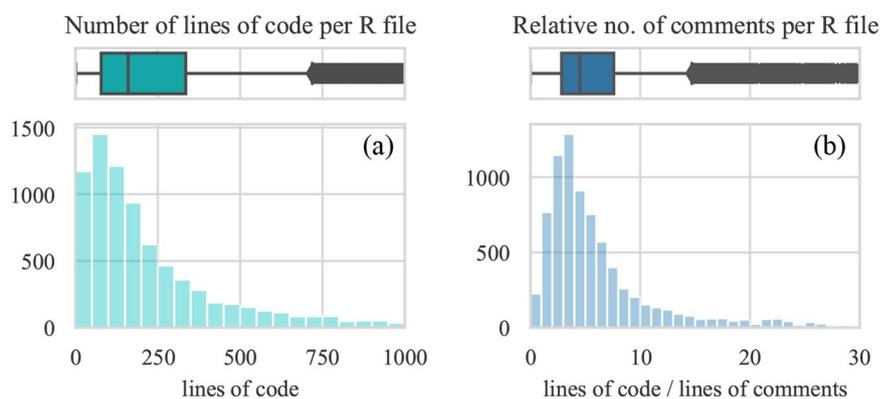


Fig. 5 Number of code line and relative number of comments.

According to IBM studies, intuitive variable naming contributes more to code readability than comments, or for that matter, any other factor¹⁴. The primary purpose of variable naming is to describe its use and content, therefore, they should not be single characters or acronyms but words or phrases. For this study, we extracted variable names from the code using the built-in R function `ls()`. Out of 3070 R files, we find that 621 use variables that are one or two characters long. However, the average length of variable names is 10, which is a positive finding as such a name could contain one, two, or more English words and be sufficiently descriptive to a reuser.

Modular programming is an approach where the code is divided into sections or modules that execute one aspect of its functionality. Each module can then be debugged, understood, and reused independently. In R programming, these modules can be implemented as functions or classes. We count the occurrences of user-defined functions and classes in R files to learn how researchers structure their code.

Out of 8875 R files, 2934 files have either functions or classes. Applying a relative number of modules per lines of code, we can estimate that one function on average contains 82 lines of code (mode is 55). According to a synthesis of interviews with top software engineers, a function should include about 10 lines of code¹². However, as noted in Section 0, R behaves like a command language in many ways, which does not inherently require users to create modules, such as functions and classes. Along these same lines, R programmers do not usually refer to their code collectively as a program but rather as a script, and it is often not written with reuse in mind.

RQ 3. What are the most used R libraries? The number of code dependencies affects the chances of reusing the code, as all dependencies (of adequate versions) need to be present for its successful execution. Therefore, a higher number of dependencies can lower the chances of their successful installation and ultimately code re-execution and reuse. We find that most datasets explicitly depend on up to 10 external libraries (Fig. 6) with individual R files requiring an average of 4.3 external dependencies (i.e., R libraries). The dependencies in code were detected by looking for “library”, “install.packages” and “require” functions.

The list of used libraries provides insight into the goals of research code (Fig. 7). Across all of the datasets, the most frequently used library is `ggplot2` for plotting, indicating that the most common task is data visualization. Another notable library is `xtable`, which offers functions for managing and displaying data in a tabular form, and similarly provides data visualization. Many libraries among the top ten are used to import and manage data, such as `foreign`, `dplyr`, `plyr` and `reshape2`. Finally, some of them are used for statistical analysis, like `stargazer`, `MASS`, `lmtest` and `car`. These libraries represent the core activities in R: managing and

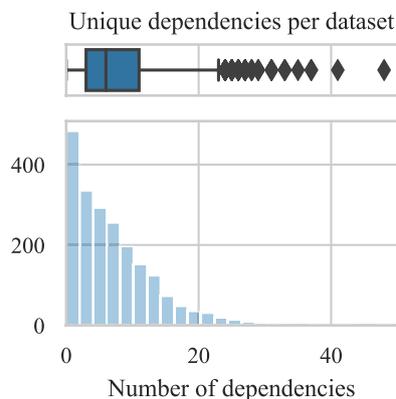


Fig. 6 Distribution of unique dependencies per dataset.

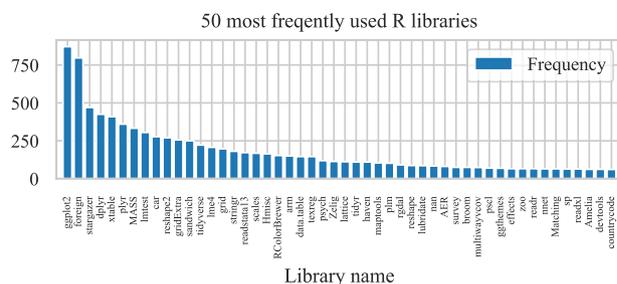


Fig. 7 Most frequently used R libraries.

formatting data, analyzing it, and producing visualizations and tables to communicate results. The preservation of these packages is, therefore, crucial for reproducibility efforts.

The infrequent usage and absence of libraries also tell us what researchers are not doing in their projects. In particular, libraries that are used for code testing, such as `runit`, `testthat`, `tinytest` and `unitizer`, were not present. Although these libraries are primarily used to test other libraries, they could also confirm that data analysis code works as expected. For example, tests of user-defined functions, such as data import or figure rendering, could be implemented using these libraries. Another approach that can aid in result validation and facilitate reproducibility is computational provenance^{17,18}. It refers to tracking data transformations with specialized R libraries, such as `provR`, `provenance`, `RDTlite`, `provTraceR`. However, in our study, we have not found a single use of these provenance libraries. In addition, libraries for runtime environment and workflow management are also notably absent. Libraries such as `renv`, `packrat` and `pacman` aid in the runtime environment management, and workflow libraries, such as `workflowR`, `workflows`, and `drake` provide explicit methods for reproducibility and workflow optimization (e.g., via caching and resource scaling). None of these were detected in our dataset, except for `renv` which was used in 2 replication packages. Though we can conclude that these approaches are not currently intuitive for the researchers, encouraging their use could significantly improve research reproducibility and reuse.

Finally, we look for configuration files used to build a runtime environment and install code dependencies. One of the common examples of a Python configuration file is `requirements.txt`, though several other options exist. For a research package (compendium), a `DESCRIPTION` configuration file that captures project metadata and dependencies was proposed¹⁹. We find 9 out of 2091 replication packages that have a `DESCRIPTION` file and 30 where the word description is contained in any of the file names. Another more Python-ic approach to the environment capture is saving dependencies in a configuration file named `install.R`. We have not found a single `install.R` among the analyzed datasets, but we found similar files such as: `install-requirements.R`, `000install.R`, `packageinstallation.R` or `postinstall`. Our results suggest that the research community that uses Harvard Dataverse does not comply with these conventions, and one reason for that may be its recent emergence as the publication was released in 2018 (before that, a `DESCRIPTION` file was typically used for R libraries). Finally, a R user may use the library `packrat` or `renv` or a built-in function `sessionInfo()` to capture the local environment. We have not found any files named `'sessionInfo'`, but we found four packages containing `.lock` files (two of those called `renv.lock`) used in the `renv` library.

RQ 4. What is the code re-execution rate? We re-executed R code from each of the replication packages using three R software versions, R 3.2, R 3.6, and R 4.0, in a clean environment. The possible re-execution outcomes for each file can be a “success”, an error, and a time limit exceeded (TLE). TLE occurs when the time allocated for file re-execution is exceeded. We allocated up to 5 hours of execution time to each replication package,

Outcomes			Combined result
Success	Anything	Anything	Success
TLE	Not Success	Not Success	TLE
Error	Error	Error	Error

Table 1. Obtaining combined re-execution result per R file from results using three versions of R software. TLE means “time limit exceeded”.

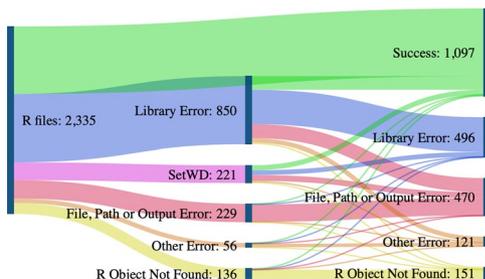


Fig. 8 Success rate and errors before and after code cleaning. To objectively determine the effects of code cleaning, we subset the results that have explicit “successes” and errors while excluding the ones with TLE values as the outcome. As a result, the count of files in this figure is lower than the total count.

and within that time, we allocated up to 1 hour to each R file. The execution time may include installing libraries or external data download if those are specified in the code. The replication packages that have resulted in TLE were excluded from the study, as they may have eventually executed properly with more time (some would take days or weeks). To analyze the success rate of the analysis, we interpret and combine the results from the three R versions in the following manner (illustrated in Table 1):

1. If there is a “success” one or more times, we consider the re-execution to be successful. In practice, this means that we have identified a version of R able to re-execute a given R file.
2. If we have one or more “TLE” and no “success”, the combined result is a TLE. The file is then excluded to avoid misclassifying a script that may have executed if given more time.
3. Finally, if we have an “error” 3 times, we consider the combined results to be an error.

We re-execute R code in two different runs, with and without code cleaning, using in each three different R software versions. We note that while 9078 R files were detected in 2109 datasets, not all of them got assigned a result. Sometimes R files exceeded the allocated time, leaving no time for the rest of the files to execute. When we combine the results based on the Table 1, we get the following results in each of the runs:

	Without code cleaning	With code cleaning	Best of both
Success rate	25%	40%	56%
Success	952	1472	1581
Error	2878	2223	1238
TLE	3829	3719	5790
Total files	7659	7414	8609
Total datasets	2071	2085	2109

Going forward, we consider the results with code cleaning as primary and further analyze them unless different is stated.

RQ 5. Can automatic code cleaning with small changes to the code aid in its re-execution? To determine the effects of the code cleaning algorithm (described in Section 4), we first re-execute original researchers’ code in a clean environment. Second, we re-execute the code after it was modified in the code-cleaning step. We find an increase in the success rate for all R versions, with a total increase of about 10% in the combined results (where only explicit errors and successes were recorded, and TLE values were excluded).

Looking at the breakdown of coding errors in Fig. 8, we see that code cleaning can successfully address some errors. In particular, it fixed all errors related to the command `setwd` that sets a working directory and is commonly used in R. Another significant jump in the re-execution rate results from resolving the errors that relate to the used libraries. Our code cleaning algorithm does not pre-install the detected libraries but instead modifies the code to check if a required library is present and installs it if it is not (see Appendix 3).

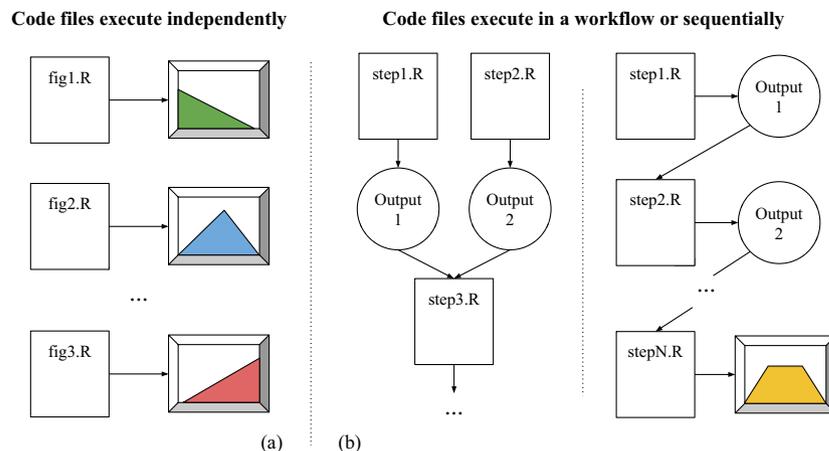


Fig. 9 Types of workflows in research analyses.

Most code files had other, more complex errors after code cleaning resolved the initial ones. For example, other library errors appeared if a library was not installed or was incompatible due to its version. Such an outcome demonstrates the need to capture the R software and dependency versions required for reuse. File, path, and output errors often appeared if the directory structure was inaccurate or if the output file was not saved. “R object not found” error occurs when using a variable that does not exist. While it is hard to pin the cause of this error precisely, it is often related to missing files or incomplete code. Due to the increased success rate with code cleaning, we note that many common errors could be avoided. There were no cases of code cleaning “breaking” the previously successful code, meaning that a simple code cleaning algorithm, such as this one, can improve code re-execution. Based on our results, we give recommendations in Section 4.

RQ 6. Are code files designed to be independent of each other or part of a workflow? R files in many datasets are designed to produce output independently of each other (Fig. 9a), while some are structured in a workflow (Fig. 9b), meaning that the files need to be executed in a specific order to produce a result. Due to the wide variety of file naming conventions, we are unable to detect the order in which the files should be executed. As a result, we may run the first step of the workflow last in the worst case, meaning that only one file (the first step) will run successfully in our re-execution study.

To examine the nature of R analysis, we aggregate the collected re-execution results in the following fashion. If there are one or more files that successfully re-executed in a dataset, we mark that dataset as ‘success’. A dataset that only contains errors is marked as ‘error’, and datasets with TLE values are removed. In these aggregated results (dataset-level), 45% of the datasets (648 out of 1447) have at least one automatically re-executable R file. There is no drastic difference between the file-level success rate (40%) and the dataset-level success rate (45%), suggesting that the majority of files in a dataset are meant to run independently. However, the success rate would likely be better had we known the execution order.

If we exclude all datasets that contain code in other programming languages, the file-level success rate is 38% (out of 2483 files), and the dataset-level success rate is 45% (out of 928 datasets). These ratios are comparable to the ones in the whole dataset (40% on file-level and 45% on dataset-level), meaning that “other code” does not significantly change the re-execution success rate. In other words, we would expect to see a lower success rate if an R file depends on the execution of the code in other languages. Such a result corroborates the assumption that R files were likely designed to be re-executed independently in most cases.

RQ 7. What is the success rate in the datasets belonging to journal Dataverse collections? More than 80 academic journals have their dedicated data collections within the Harvard Dataverse repository to support data and code sharing as supplementary material to a publication. Most of these journals require or encourage researchers to release their data, code, and other material upon publication to enable research verification and reproducibility. By selecting the datasets linked to a journal, we find a slightly higher than average re-execution rate (42% and aggregated 47% instead of 40% and 45%). We examine the data further to see if a journal data sharing policy influences its re-execution rate.

We survey data sharing policies for a selection of journals and classify them into five categories according to whether data sharing is: encouraged, required, reviewed, verified, or there is no policy. We analyze only the journals with more than 30 datasets in their Dataverse collections. Figure 10 incorporates the survey and the re-execution results. “No-policy” means that journals do not mandate the release of datasets. “Encouraged” means that journals suggest to authors to make their datasets available. “Required” journals mandate that authors make their dataset available. “Reviewed” journals make datasets part of their review process and ensure that it plays a role in the acceptance decision. For example, the journal Political Analysis (PA) provides detailed instruction on what should be made available in a dataset and conducts “completeness reviews” to ensure published datasets meet those requirements. Finally, “verified” means the journals ensure that the datasets enable reproducing the results presented in a paper. For example, the American Journal of Political Science (AJPS)

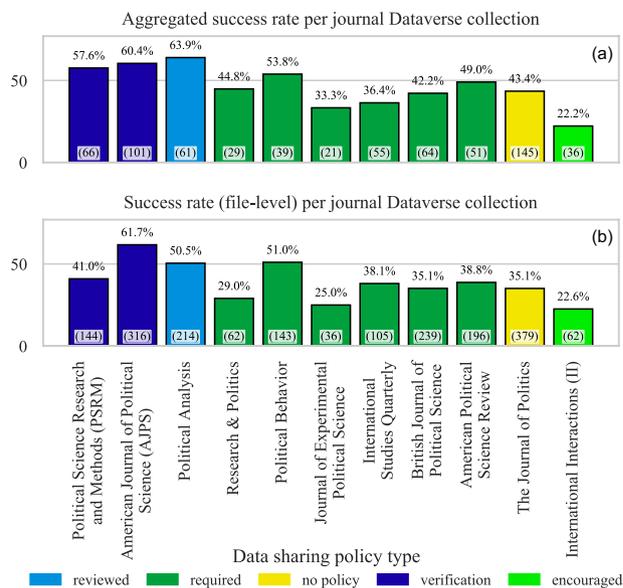


Fig. 10 Re-execution success rate per journal Dataverse collection. In the brackets are the number of datasets and the number of R files, respectively.

Publication year	2010	2014	2015	2016	2017	2018	2019	2020
Dataset count	1	2	90	284	335	493	555	241

Table 2. Dataset publication date.

R version	Release date
R 3.2.1	June, 2015
R 3.6.0	April, 2019
R 4.0.1	June, 2020

Table 3. Release date of used R versions.

requires authors to provide all the research material necessary to support the paper claims. Upon acceptance, the research material submitted by authors is verified to ensure that they produce the reported results^{20,21}. From Fig. 10 we see that the journals with the strictest policies (Political Science Research and Methods, AJPS, and PA) have the highest re-execution rates (Fig. 10a shows aggregated result per dataset, and Fig. 10b shows file-level results). Therefore, our results suggest that the strictness of the data sharing policy is positively correlated to the re-execution rate of code files.

RQ 8. How do release dates of R and datasets impact the re-execution rate? The datasets in our study were published from 2010 to July 2020 (Table 2), which gives us a unique perspective in exploring how the passing of time affects the code. In particular, R libraries are not often developed with backward compatibility, meaning that using a different version from the one used originally might cause errors when re-executing the code. Furthermore, in some cases, the R software might not be compatible with some versions of the libraries.

Considering the release years of R versions (Table 3), we examine the correlation in the success rates between them and the release year of the replication package. We find that R 3.2, released in 2015, performed best with the replication packages released in 2016 and 2017 (Fig. 11a). Such a result is expected because these replication packages were likely developed in 2015, 2016, and 2017 when R 3.2 was frequently used. We also see that it has had a lower success rate in recent years. We observe that R 3.6 has the highest success rate per year (Fig. 11b). This R version likely had some backward compatibility with older R subversions, which explains its high success rate in 2016 and 2017. Lastly, R 4.0 is a recent version representing a significant change in the software, which explains its generally low success rate (Fig. 11c). Because R 4.0 was released in summer 2020, likely none of the examined replication packages originally used that version of the software. It is important to note that the subversion R 3.6 was the last before the R 4.0 (i.e., there was no R 3.7 or later subversions). All in all, though we see some evidence of backward compatibility, we do not find a significant correlation between the R version and the release year of a replication package. A potential cause may be the use of incompatible library versions in our re-execution step as the R software automatically installs the latest version of a library. In any case, our result

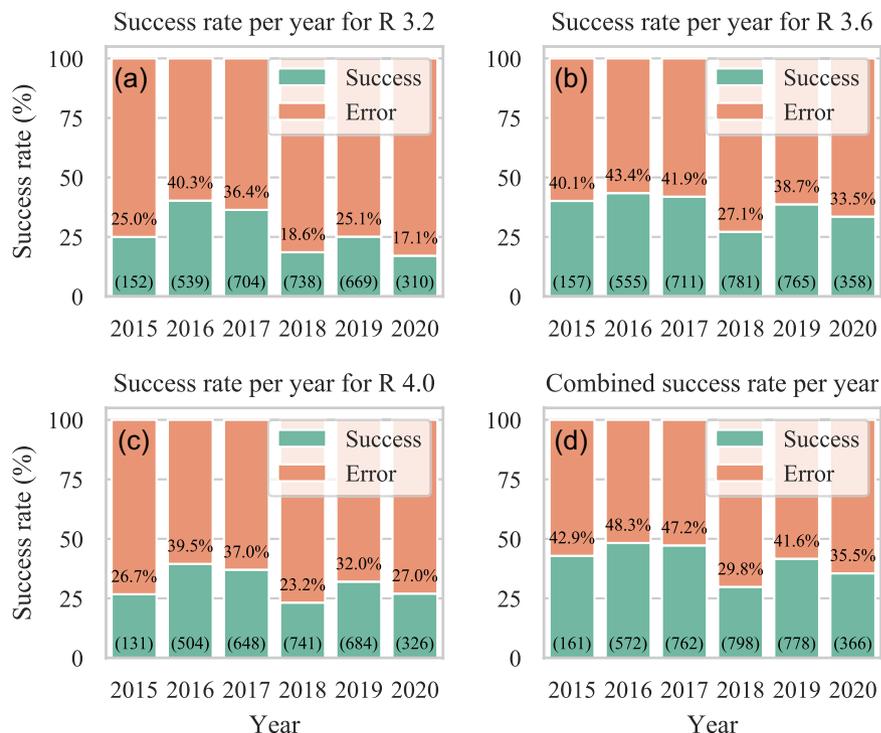


Fig. 11 Re-execution success rates per year per R software version.

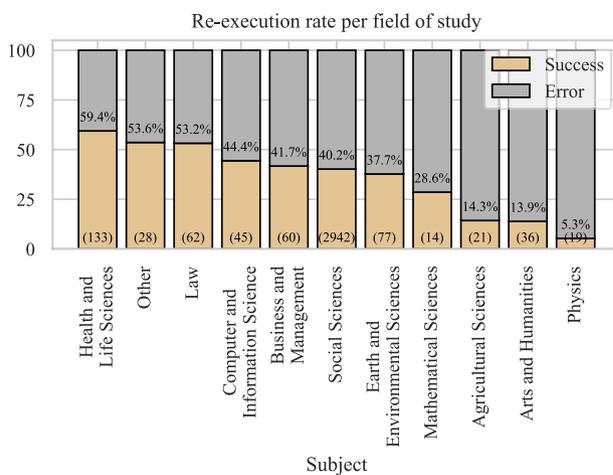


Fig. 12 Success rate per research field.

highlights that the execution environment evolves over time and that additional effort is needed to ensure that one can successfully recreate it for reuse.

Looking at the combined result from Fig. 11d, we are unable to draw significant conclusions on whether the old code had more or fewer errors compared to the recent code (especially considering the sample size per year). Similarly, from 2015 to 2020, we do not observe significant changes in the dataset size, the number of files in the dataset, nor the number of R code files, which remain around reported values (see RQ 1). However, we see that the datasets use an increasing number of (unique) libraries over time, i.e., from an average of 6 in 2015 to about 9 in 2020.

RQ 9. What is the success rate per research field? The Harvard Dataverse repository was initially geared toward social science, but it has since become a multi-disciplinary research data repository. Still, most of the datasets are labeled 'social science,' though some have multiple subject labels. To avoid sorting the same dataset into multiple fields, if a dataset was labeled both "social science" and "law," we would keep only the latter. In other words, we favored a more specific field (such as "law") and chose it over a general one (like "social science") when that was possible.

Outcomes	Dataset count
Only success	127
Only error	654
Only TLE	684
Success & error	167
Success & TLE	215
Error & TLE	143
Success, error & TLE	114
Total	2085

Table 4. Re-execution results combination per dataset. For example, there are 215 datasets that contain only 'success' and 'TLE' as outputs.

The re-execution rates per field of study are shown in Fig. 12. The highest re-execution rates were observed for the health and life sciences. It may be that medical-related fields have a stronger level of proof embedded in their research culture. Physics had the lowest re-execution rate, though due to the low sample size and the fact that Dataverse is probably not the repository of choice in physics, we cannot draw conclusions about the field. Similarly, we cannot draw conclusions for many of the other fields due to the low sample size (see the number of R files in the brackets), and therefore, the results should not be generalized.

RQ 10. How does the re-execution rate relate to research reproducibility? Research is reproducible if the shared data and code produce the same output as the reported one. Code re-execution is one of the essential aspects of its quality and a prerequisite for computational reproducibility. However, even when the code re-executes successfully in our study, it does not mean that it produced the reported results. To access how the code re-execution relates to research reproducibility, we select a random sample of three datasets where all files were executed successfully and attempt to compare its outputs to the reported ones. There are 127 datasets where all R files re-executed with success (Table 4).

The first dataset from the random sample is a replication package linked to a published paper at the Journal of Experimental Political Science²². It contains three R files and a Readme, among other files. The Readme explains that each of the R files represents a separate study and that the code logs are available within the dataset. Comparing the logs before and after code re-execution would be a good indication of its success. After re-executing two of the R files, we find that the log files are almost identical and contain identical tables. The recreated third log file nearly matched the original, but there were occasional discrepancies in some of the decimal digits (though the outputs were in the same order of magnitude). Re-executing the third R file produced a warning that the library `SDMTtools` is not available for R 3.6, which may have caused the discrepancy.

The second dataset from the random sample is a replication package linked to a paper published at Research & Politics²³. It has a single R file and a Readme explaining that the script produces a correlation plot. According to our results, it should be re-executed with R 3.2. The R file prints two correlation coefficients, but it is unable to save the plot in the Docker container.

The final dataset from our sample follows a paper published at the Review of Economics and Statistics²⁴. It contains two R files and a Readme. One of the R files contains only functions, while the other calls the functions file. Running the main R file prints a series of numbers, which are estimates and probabilities specified in the document. However, the pop-up plotting functions are suppressed due to the re-execution in the Docker container. The code does not give errors or warnings.

While successful code re-execution is not a sufficient measure for reproducibility, our sample suggests that it might be a good indicator that computational reproducibility will be successful.

Limitations of the Study

Though this study is framed around pre-defined research questions, it has certain limitations in answering them due to its automatized and large-scale nature, which cannot detect the nuances in research code. For instance, code re-execution is a necessary but not sufficient requirement for reproducibility. Re-executable code may produce a different result than the reported one and be a "false positive" in our dataset. However, it is likely that small changes in the re-execution would also result in a higher success rate. In some cases, the code would not fail to complete if it was re-executed in RStudio or had a minor change in file paths. In particular, we observe numerous errors related to directory paths, ranging from system encoding to accessing unavailable file directories. The issue is partially solved with the use of `basename` function in the code cleaning stage. All things considered, we note that our automated study has a comparable success rate to the reported manual reproducibility studies^{25,26}, which gives strength to the overall significance of our results. Though conducting a reproducibility study with human intervention would result in more sophisticated findings, it is labor-intensive on a large scale. We should strive toward enabling reproducibility studies with automation while using existing standards like machine-readable FAIR data²⁷ and code^{28,29}, and this paper provides recommendations toward that goal.

Even though it may appear that we should have tested the code with a higher number of R software versions, we believe that our results would not have been drastically different. Indeed this is a limitation of all R software versions, as each would by default try to install the latest version of R libraries. As a result, the research code

would fail if the R version and the library version were not compatible. Even if the library was successfully installed, it might not work as expected if the author used an earlier version.

Some of the limitations were imposed by the number of resources we had on the AWS cloud. For instance, there is a large count of TLE values in our results even though we set the time limitation to be 1 hour per R file and 5 hours per dataset. The main cause of depleting the AWS resources was a choice of a large EC2 instance for the re-execution. In hindsight, these instances were excessive for this type of study, and we should have used small to medium-sized instances.

Finally, one might consider a limitation that we use datasets from a single source. While code repositories such as GitHub are used for software development, temporary projects, research, education, and more, the datasets published on Harvard Dataverse are primarily created for research purposes. Furthermore, the data curation team at Harvard vets all published datasets to maintain its data and research focus. Therefore, this study provides an insight into researchers' coding practices in R, and for that reason, it required a research-focused repository such as Dataverse.

Best Practices and Recommendations

In this study, we saw that many errors in research code could be prevented with basic good practices. Extensive guides on code practices have been published^{30–34} and are available online³⁵. However, based on our results, we provide the following core recommendations for the researchers who use R:

1. Library versions in a project should be captured ideally by using the `renv` package. Otherwise, you may use a `DESCRIPTION` file or `install.R` or include the output of `sessionInfo()` from the researcher's R session.
2. Use relative file paths in your code. Absolute (or full) file paths are a frequent cause of error when re-executing code on a new file system.
3. Workflow capture and management methods such as R Markdown, `drake`, and its successor, `targets` will help to automate your code and specify the correct execution sequence.
4. Use Docker to document your runtime environment in a machine-readable way, and to ensure others can recreate your computational environment with all the necessary dependencies. For instance, the Rocker Project³⁶ (<https://www.rocker-project.org>) provides widely used Docker images for R across different applications.
5. Test your code in a clean environment before sharing or publishing it, as it could help you identify dependencies and missing files.
6. Use free and open-source software whenever possible, as proprietary software can hinder transparency and reproducibility of your research.

Data and software repositories aim to provide high-quality resources that can be leveraged for further research. The following are our recommendations for data repository operators and curators:

1. Capturing re-execution commands for each research dataset would be immensely helpful for reusers. It would resolve the ambiguity of the file execution order (workflow) and showcase its input arguments. Support for metadata fields or files capturing such commands does not currently exist but could be incorporated into dataset metadata at the repository.
2. Data repository integration with reproducibility platforms (such as Code Ocean^{37,38}, Whole Tale³⁹, Jupyter Binder⁴⁰) and Renku (<https://datascience.ch/renku>) could help capture library dependencies and test the code before it is published. It could be implemented as a part of the research submission workflow⁴¹. These tools use container technology that has been deemed valuable for preserving code⁴².
3. Creating a working group that would support various aspects of reproducible research, investigate state of the art, and improve the quality of shared code would be beneficial. At Dataverse, we have created a *Software, Workflows, and Containers* working group, which gathers experts, identifies community-wide problems, prioritizes them, and implements solutions in the Dataverse software.

Finally, our results suggest that journal data policy strictness positively correlates with the observed code re-execution rate. Therefore, journals play a critical role in making scholarly communication successful, and they have the power to require that the underlying data and code accompany articles. Our recommendations for the journal editors:

1. Consider implementing a simple review of all deposited material if a code verification is infeasible for your journal.
2. Create a reproducibility checklist that include code best practices (as recommended above) and testing code re-execution in a clean environment to make the submission and review process more straightforward^{43–45}.
3. Consider recommending the use of certain libraries or tools that facilitate code automation and re-execution.

Though our study primarily aimed to identify code re-execution and quality issues, the developed AWS pipeline can find its use outside of the study boundaries. For instance, the pipeline can be transformed into a GitHub Action (<https://github.com/features/actions>) that helps researchers test their code with every new commit. As a result, execution errors would be promptly identified and fixed. Data repositories and journals could also

incorporate the pipeline at code upload on their respective platforms. That way, the code would be automatically re-executed before it reaches editors, reviewers, curators, or reusers, who could then make decisions based on the re-execution result and code analysis. However, deploying such a pipeline would require additional costs for computation and may present a hurdle for users releasing open data. Therefore, for these infrastructures, it could be viable as an optional feature rather than a submission requirement. We explore some aspects of this use in follow-up works⁴⁶.

Related Work

Claims about a reproducibility crisis attracted attention even from the popular media, and many studies on the quality and robustness of research results have been performed in the last decade^{2,3}. Most reproducibility studies were done manually, where researchers tried to reproduce previous work by following its documentation and occasionally contacting original authors. Given that most of the datasets in our study belong to the social sciences, we reference a few reproducibility studies in this domain that emphasize its computational component (i.e., use the same data and code). Chang and Li attempt to reproduce results from 67 papers published in 13 well-regarded economic journals using the deposited supplementary material²⁶. They successfully reproduced 33% of the results without contacting the authors and 43% with the authors' assistance. Some of the reasons for the reduced reproducibility rate are proprietary software and missing (or sensitive) data. Stodden and collaborators conduct a study reporting on both reproducibility rate and journal policy effectiveness²⁵. They look into 204 scientific papers published in the journal *Science*, which previously implemented a data sharing policy. The authors report being able to obtain resources from 44% of the papers and reproduce 26% of the findings. They conclude that while a policy represents an improvement, it does not suffice for reproducibility. These studies give strength to our analysis as the success rates are comparable. Furthermore, by examining multiple journals with various data policy strictness, we corroborate the finding that open data policy is an improvement but less effective than code review or verification in enabling code re-execution and reproducibility.

Studies that focus primarily on the R programming language have been reported. Konkol and collaborators conducted an online survey among geoscientists to learn of their experience in reproducing prior work⁴⁷. In addition, they conducted a reproducibility study by collecting papers that included R code and attempting to execute it. Among the 146 survey participants, 7% tried to reproduce previous results, and about a quarter of those have done that successfully. For the reproducibility part of the study, Konkol and collaborators use RStudio and a Docker image tailored to the geoscience domain. They report that two studies ran without any issues, 33 had resolvable issues, and two had issues that could not be resolved. For the 15 studies, they contacted the corresponding authors. In total, they encountered 173 issues in 39 papers. While we cannot directly compare the success rate due to the different approaches, we note that much of the reported issues overlap. In particular, issues like a wrong directory, deprecated function, missing library, missing data, and faulty calls that they report are also frequently seen in our study.

Large-scale studies have the strength to process hundreds of datasets in the same manner and examine common themes. Our study is loosely inspired by the effort undertaken by Chen⁴⁸ in his undergraduate coursework, though our implementation, code-cleaning and analysis goals differ. Pimentel and collaborators retrieved over 860,000 Jupyter notebooks from the Github code repository and analyzed their quality and reproducibility⁴⁹. The study first attempted to prepare the notebooks' Python environment, which was successful for about 788,813 notebooks. Out of those, 9,982 notebooks exceeded a time limit, while 570,476 failed due to an error. A total of 208,323 of the notebooks finished their execution successfully (24.11%). About 4% re-executed with the same result, which was inferred by comparing it with the existing outputs in the notebook. This result is comparable to the re-execution rate of 27% in our previous analysis of Python code from Harvard Dataverse repository⁶. We also note that Pimentel and collaborators performed the study on diverse Jupyter notebooks, which often include prototype development and educational coding. Our study is solely based on research code in its final (published) version. The studies are not directly comparable due to the use of different programming languages. However, we achieve a comparable result of 25% when re-executing code without code cleaning. Also, the fact that the most frequent errors relate to the libraries in both studies signals that both programming languages face similar problems in software sustainability and dependency capture.

Technical Implementation of Code Cleaning

Our code cleaning implementation aims to solve some of the most common errors and ensure that used libraries are installed. All R files are converted to ASCII (to reduce the chance of syntax error caused by symbols from other operating systems), scanned and modified if a common problem is detected. Our code cleaning approach is relatively simple to minimize the chance of "breaking the code", and we do not use static analysis packages such as `goodpractice` or `lintr` as they do not make changes in the code automatically.

While some R users set their working directory with the `setwd` function, this often causes errors if the directory path cannot be found. This is one of the commands that our code cleaning implementation targets. It detects the `setwd` function and replaces it with a new one that makes the current directory with the downloaded files, a working directory.

Other file path errors were solved with the use of the `basename` function. The `basename` function ignores the path and the path separators. For example, if a long file path is used in the `read.csv` function, it will be ignored, and only the name of the target file will be used. Such an approach works in this scenario because all files from the replication package are downloaded one by one and stored in the same directory. In the implementation, a code line like: `file.path("/Dropbox/my_datafile.csv")` would therefore be replaced with:

```
basename(file.path("/Dropbox/my_datafile.csv"))
```

In the R programming, all libraries can be installed directly from the script using the `install.packages` command. Therefore if we detect that a needed library was not present in the working environment, we should add this command to install it and avoid an error. We tested a few approaches to identify the used and pre-installed libraries. Ultimately, using a combination of the functions `require` and `install.packages` proved to be the best solution (i.e., only if the library cannot be loaded with `require` it is installed from the code). The benefit of using the “require” function is that it returns a logical value by default, or “true” if the package is loaded and “false” if it is not. Therefore, we could check if the package is present and only install it if it is not. Such an approach saved time and reduced the chances of errors caused by duplicated code. As an example from the implementation, a line `library(dplyr)` would be replaced with:

```
if(!require("dplyr")) install.packages("dplyr")
```

Usage of the `::` method in R allows access to a single function from a specific package. For instance, to use `intersect` from the `dplyr` package, one would invoke it as `dplyr::intersect`. It is important to note that the package must be installed in the environment to be available using this method, but it does not have to be loaded in the R script. Though all re-executions were conducted in an environment that included `r-essentials` (or about 200 most popular R packages), we have not implemented a pre-detection of packages that used the `::` method. We expect that including it would somewhat (likely marginally) improve the results.

The US CRAN (<http://cran.us.r-project.org>) was set as default to avoid the CRAN errors.

Data availability

The study uses several collected datasets, which are then merged into derived datasets and used for the final analysis. All data records are stored in a simple CSV (or text) file format and can be opened with a text editor. The datasets were collected in the AWS re-execution step unless otherwise indicated. The collected datasets are:

- `run_log_r[VERSION]_[NO_]env` provides information of whether or not an R code file executed with an error for three versions of R (3.2, 3.6 and 4.0) and in two different environments (before and after code cleaning step).
- `run_log_r[VERSION]_[NO_]env[_DOWNLOAD]` provides information whether all files from a replication package downloaded without a checksum error.
- `rfile_stats` gathers information on each R code file, such as the number of code lines, code comments, functions, classes, tests and file encoding.
- `rpackage_stats` gathers information on the replication package such as total comments, total libraries, size, list of files and list of used libraries.
- `readability_metrics` captures research code readability metrics such as the average number of periods, commas, spaces, parenthesis, which was explored in a follow-up publication at ref. ⁴⁶.
- `all_metadata` captures the collection (journal or other) and the academic field of each replication dataset. Retrieved from Harvard Dataverse directly.
- `journal-policy-survey` gathers information on the strictness of data sharing policies at academic journals that have their own collections on Harvard Dataverse. Collected manually.

All data records are freely available on the Harvard Dataverse repository at <https://doi.org/10.7910/DVN/UZLXSZ> under CC0 license⁹. The same copy of data files is available on GitHub at <https://github.com/atrisovic/dataverse-r-study>.

Code availability

To develop and execute the analysis code, we used Python 2.7. The code is released as a single version, which was used for both data collection and analysis. All Python dependencies with their versions are captured in a text file `requirements.txt` at the root directory. All code files can be freely accessed on on GitHub at <https://github.com/atrisovic/dataverse-r-study>. The code is released under MIT license.

Received: 26 March 2021; Accepted: 17 January 2022;

Published online: 21 February 2022

References

1. Digital Science *et al.* The State of Open Data Report 2019. *Digital Science* <https://doi.org/10.6084/m9.figshare.9980783.v2> (2019).
2. National Academies of Sciences, Engineering, and Medicine and others. *Reproducibility and replicability in science* (National Academies Press, 2019).
3. Baker, M. 1,500 scientists lift the lid on reproducibility. *Nature News* **533**, 452 (2016).
4. Knuth, D. E. Literate programming. *The Computer Journal* **27**, 97–111 (1984).
5. Pasquier, T. *et al.* Sharing and Preserving Computational Analyses for Posterity with encapsulator. *Computing in Science Engineering* **20**, 111–124 (2018).
6. Trisovic, A., Mika, K., Boyd, C., Feger, S. & Crosas, M. Repository approaches to improving the quality of shared data and code. *Data* **6**, 15 (2021).
7. Muenchen, R. A. The popularity of data analysis software. Preprint at <http://r4stats.com/popularity> (2012).
8. Kaggle ML & DS Survey. The most comprehensive dataset available on the state of ML and data science <https://www.kaggle.com/kaggle/kaggle-survey-2018> (2018).
9. Trisovic, A. Replication Data for: A large-scale study on research code quality and execution. *Harvard Dataverse* <https://doi.org/10.7910/DVN/UZLXSZ> (2021).
10. Baumer, B., Cetinkaya-Rundel, M., Bray, A., Loi, L. & Horton, N. J. R markdown: Integrating a reproducible analysis tool into introductory statistics. Preprint at <https://arxiv.org/abs/1402.1894> (2014).
11. W3C Working Group. Best practices for publishing linked data <http://www.w3.org/TR/ld-bp/> (2014).

12. Martin, R. C. *Clean code: a handbook of agile software craftsmanship* (Pearson Education, 2009).
13. Sandve, G. K., Nekrutenko, A., Taylor, J. & Hovig, E. Ten Simple Rules for Reproducible Computational Research. *PLoS Comput Biol* **9** (Public Library of Science, 2013).
14. McConnell, S. *Code complete* (Pearson Education, 2004).
15. Thomas, D. & Hunt, A. *The Pragmatic Programmer: your journey to mastery* (Addison-Wesley Professional, 2019).
16. Hyde, R. *Write great code: Thinking low-level, writing high-level*, **2** (No Starch Press, 2020).
17. Davidson, S. B. & Freire, J. Provenance and scientific workflows: challenges and opportunities. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 1345–1350 (2008).
18. Pasquier, T. *et al.* If these data could talk. *Scientific data* **4**, 1–5 (2017).
19. Marwick, B., Boettiger, C. & Mullen, L. Packaging data analytical work reproducibly using R (and friends). *The American Statistician* **72**, 80–88 (2018).
20. Vines, T. H. *et al.* Mandated data archiving greatly improves access to research data. *The FASEB Journal* **27**, 1304–1308 (2013).
21. Crosas, M. *et al.* Data policies of highly-ranked social science journals. Preprint at <https://doi.org/10.31235/osf.io/9h7ay> (2018).
22. White, A. & Trump, K.-S. Replication Data for: Does Inequality Beget Inequality? Experimental tests of the prediction that inequality increases system justification motivation. *Harvard Dataverse* <https://doi.org/10.7910/DVN/SWV9GJ> (2017).
23. Polk, J. Replication Data for: Explaining the Salience of Anti-Elitism and Reducing Political Corruption for Political Parties in Europe with the 2014 Chapel Hill Expert Survey Data. *Harvard Dataverse* <https://doi.org/10.7910/DVN/Z02C8Y> (2019).
24. Helland, E. & Yoon, J. Replication Data for: Estimating the Effects of the English Rule on Litigation Outcomes. *Harvard Dataverse* <https://doi.org/10.7910/DVN/YT45AO> (2016).
25. Stodden, V., Seiler, J. & Ma, Z. An empirical analysis of journal policy effectiveness for computational reproducibility. *Proceedings of the National Academy of Sciences* **115**, 2584–2589 (National Academy of Sciences, 2018).
26. Chang, A. C. & Li, P. Is economics research replicable? Sixty published papers from thirteen journals say “usually not”. *Finance and Economics Discussion Series 2015-083* (2015).
27. Wilkinson, M. D. *et al.* The FAIR guiding principles for scientific data management and stewardship. *Scientific data* **3**, 1–9 (2016).
28. Lamprecht, A.-L. *et al.* Towards fair principles for research software. *Data Science* **3**, 37–59 (2020).
29. Jiménez, R. C. *et al.* Four simple recommendations to encourage best practices in research software. *F1000Research* **6** (2017).
30. Sayre, F. & Riegelman, A. Replicable services for reproducible research: a model for academic libraries. *College & Research Libraries* **80**, 260 (2019).
31. Sandve, G. K., Nekrutenko, A., Taylor, J. & Hovig, E. Ten simple rules for reproducible computational research. *PLoS Comput Biol* **9** (2013).
32. Chen, X. *et al.* Open is not enough. *Nature Physics* **15**, 113–119 (2019).
33. Lee, B. D. Ten simple rules for documenting scientific software. *PLoS Comput Biol* **14**, e1006561 (2018).
34. Barba, L. A. Praxis of reproducible computational science. *Computing in Science & Engineering* **21**, 73–78 (2019).
35. Arnold, B. *et al.* The turing way: A handbook for reproducible data science <https://the-turing-way.netlify.app> (2019).
36. Nüst, D. *et al.* The Rockerverse: Packages and Applications for Containerisation with R. *The R Journal* **12**, 437–461 (2020).
37. Clyburne-Sherin, A., Fei, X. & Green, S. A. Computational reproducibility via containers in psychology. *Meta-psychology* **3** (2019).
38. Cheifet, B. Promoting reproducibility with Code Ocean. *Genome Biol* **22**, 65 (2021).
39. Brinckman, A. *et al.* Computing environments for reproducibility: Capturing the “whole tale”. *Future Generation Computer Systems* **94**, 854–867 (2019).
40. Jupyter *et al.* Binder 2.0 - reproducible, interactive, sharable environments for science at scale (2018).
41. Trisovic, A. *et al.* Advancing Computational Reproducibility in the Dataverse Data Repository Platform. *Proceedings of the 3rd International Workshop on Practical Reproducible Evaluation of Computer Systems*, 15–20 (ACM, 2020).
42. Chuah, J., Deeds, M., Malik, T., Choi, Y. & Goodall, J. L. Documenting computing environments for reproducible experiments. *PARCO*, 756–765 (2019).
43. The Software Sustainability Institute. Checklist for a software management plan. *Zenodo* <https://doi.org/10.5281/zenodo.2159713> (2018).
44. Goeva, A., Stoudt, S. & Trisovic, A. Toward reproducible and extensible research: from values to action. *Harvard Data Science Review* **2** (2020).
45. Pineau, J. *et al.* Improving reproducibility in machine learning research: a report from the NeurIPS 2019 reproducibility program. *Journal of Machine Learning Research* **22** (2021).
46. Bahaidarah, L. *et al.* Toward reusable science with readable code and reproducibility. Preprint at <https://arxiv.org/abs/2109.10387> (2021).
47. Konkol, M., Kray, C. & Pfeiffer, M. Computational reproducibility in geoscientific papers: Insights from a series of studies with geoscientists and a reproduction study. *International Journal of Geographical Information Science* **33**, 408–429 (2019).
48. Chen, C. *Coding BetteR: Assessing and Improving the Reproducibility of R-Based Research With containR*. Preprint at <https://dash.harvard.edu/handle/1/38811561> (2018).
49. Pimentel, J. F., Murta, L., Braganholo, V. & Freire, J. A large-scale study about quality and reproducibility of Jupyter notebooks. *IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 507–517 (IEEE, 2019).

Acknowledgements

This work is partially funded by the Sloan Foundation (award number G-2018-11111). A. Trisovic is funded by the Alfred P. Sloan Foundation (grant number P-2020-13988). This work was supported by the AWS Research Credits. A. T. thanks Christopher Chen, Steven Worthington, Leonid Andreev and Åsmund S. Folkestad for helpful conversations. The authors thank Christopher Chen and Margo Seltzer for their contributions. Thank you to the Dataverse team at Harvard’s IQSS for their help and support.

Author contributions

A.T.: conceptualization, software, formal analysis, investigation, writing– original draft preparation, methodology, validation, visualization, writing– review and editing, funding acquisition, data curation M.K.L.: conceptualization, writing–review and editing T.P.: conceptualization, writing–review and editing M.C.: conceptualization, supervision, writing–review and editing, project administration, funding acquisition.

Competing interests

The authors declare no competing interests.

Additional information

Correspondence and requests for materials should be addressed to A.T.

Reprints and permissions information is available at www.nature.com/reprints.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/>.

© The Author(s) 2022