Addressing The Scientific Reproducibility Crisis Through Educational Software
Integration

Audrey M. Bertin

Acknowledgements

Table of Contents

List of Figures

Abstract

The preface pretty much says it all.

Second paragraph of abstract starts here.

Introduction

If it weren't for Professor Ben Baumer, I would not be here writing this thesis today. The Introduction to Data Science course I took with him in my first semester at Smith College spurred on my love for the field, prompting me to declare a major—with Ben as my advisor—soon after.

That course—SDS 192—was my first introduction to the topic of scientific reproducibility. In Ben's class, one of the first undergraduate courses in the country to implement reproducibility as part of the curriculum, I was introduced to a variety of tools that could help make my work more reproducible. I learned how to write effective narrative reports with RMarkdown, how to implement version control on GitHub, how to write well-styled code, and how to use R projects to structure my files in a way that was easy to manage.

Through this work, I began to see the benefits of emphasizing a reproducible workflow. When I worked on projects with other students, the tools I learned in class enabled us to collaborate with one another easily and keep track of the progress we had made. And when I tried to look back at my own work from the past, my attention to reproducibility meant that it was much easier for me to re-compute and understand what I had done previously.

Additionally, my work in Introduction to Data Science helped build a strong interest in R. I developed a love for programming in the language and was curious to learn more about how it worked—both behind the scenes and practice. This interest prompted me to pursue research in the field. I wanted to expand my knowledge and challenge myself by learning about and solving cutting-edge issues in the data science domain.

As soon as I felt that I had built up enough coding experience to be able to tackle challenging issues, I reached out to the data science faculty to see if there were any opportunities. Ben enthusiastically welcomed me in, telling me that he had recently started developing an R package focused on reproducibility and offering me a position assisting him with writing it. That was when I first learned about `fertile`.

As part of his interest in reproducibility, Ben had been working—with help from Hadley Wickham and Jenny Bryan of RStudio—to develop a package designed to help users create optimal conditions for reproducible work in R (titled "fertile").

When it was presented to me, the package was incomplete. Several of the major functions that still exist now were there at the time, but there were not many features beyond the basics. Additionally, several functions were not fully written or throwing errors, the package had little to no documentation, and the majority of tests that had

been written to test the functions were failing.

I was given the opportunity to join in on the project and help move `fertile` to a functional state: to fix the errors that were present, document the package, and write new functions to expand the operation beyond the barebones structure that was present.

The `fertile` project was an excellent fit for me. The package was not yet in a functional state, but had an excellent base of code to work off of. Given that it already had some basic functions and a strong structure, `fertile` provided a perfect opportunity for me to learn about the process of writing R packages.

The more I worked on `fertile`, the more interested in the project I became. The package was unlike anything else I had been exposed to in data science. Most of my course work had been in the form of front-end coding: conducting data analyses and visualizing information. `fertile` is nothing like that. It operates completely behind the scenes, conducting a kind of meta-analysis to determine the reproducibility of a data science project. Rather than writing code to analyze data, I had to learn how to write code to analyze code that *other people* wrote to analyze data.

This was an incredibly complex problem. It involved capturing information about the user's file system and their RStudio environment, both of which I knew nothing about. It also involved tracking and recording user behavior—something defniitely not taught in data analysis courses. Even experienced R users deemed some of the goals associated with the package to be incredibly lofty. On the question of whether it was possible to test whether R code contained uncontrolled randomness, one coder said it was "probably impossible", while another said there was "fundamentally no way" to achieve it.

The challenge of the problems at hand, however, was what made them interesting. Most of the time, there was no way to use my past coding knowledge to solve them— rather, I had to research and test out new solutions. While the process was at times incredibly frustrating, particularly given that some of the errors I came across had likely never been seen by any other R user, it was an incredible learning opportunity. I was exposed to a wide variety of R features I would likely have never seen otherwise and greatly broadened my problem-solving and troubleshooting skills.

Every problem I solved made me more confident in my abilities and more excited to work on the `fertile`. I delved more into the topic of reproducibility and looked at some of the work by other researchers and code developers. This research introduced me to new aspects of reproducibility I wanted to include in the package and inspired me to expand its scope.

This paper is the culmination of the work I have been doing with `fertile` for the past two years.

Chapter 1

An Introduction to Reproducibility

1.1   What Is Reproducibility?

In the field of data science, research is considered fully *reproducible* when the requisite code and data files produce identical results when run by another analyst, or more generally, when a researcher can "duplicate the results of a prior study using the same materials as were used by the original investigator" (Bollen et al. (2015)).

This term was first coined in 1992 by computer scientist Jon Claerbout, who associated it with a "software platform and set of procedures that permit the reader of a paper to see the entire processing trail from the raw data and code to figures and tables" (Claerbout & Karrenbach (1992)).

Since its inception, the concept of reproducibility has been applied across many different data-intensive fields, including epidemiology, computational biology, economics, clinical trials, and, now, the more general domain of statistical and data sciences (Goodman, Fanelli, & Ioannidis (2016)).

Reproducible research has a wide variety of benefits in the scientific community. When researchers provide the code and data used for their work in a well-organized and reproducible format, readers are more easily able to determine the veracity of any findings by following the steps from raw data to conclusions. The creators of reproducible research can also more easily receive more specific feedback (including bug fixes) on their work. Moreover, others interested in the research topic can use the code to apply the methods and ideas used in one project to their own work with minimal effort.

Although often confused, the concept of *reproducibility* is distinct from the similar idea of *replicability*: the ability of a researcher to duplicate the results of a study when following the original procedure but collecting new data. Replicability has larger-scale implications than reproducibilty; the findings of research studies can not be accepted unless a variety of other researchers come to the same conclusions through independent work.
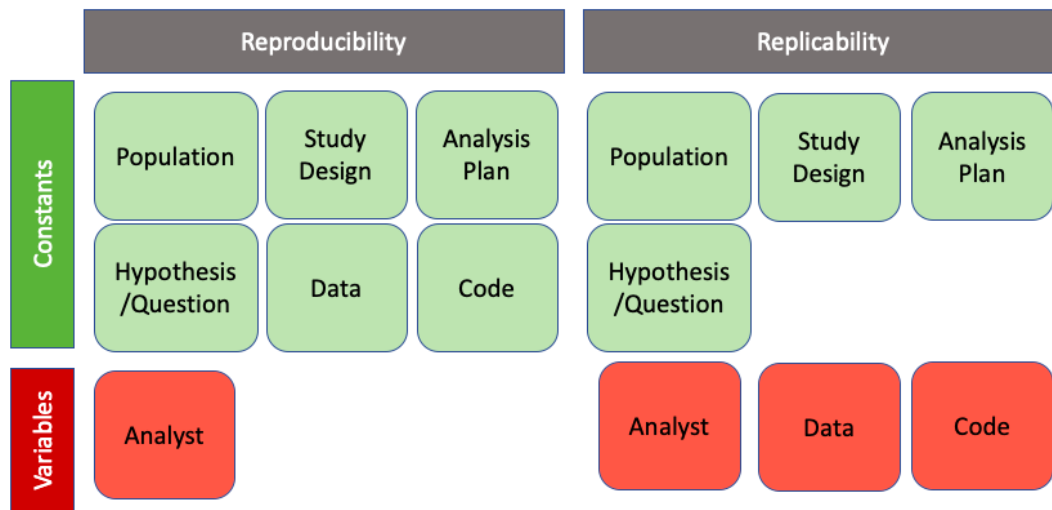
Figure 1.1: Reproducibility vs. Replicability

Reproducibility and replicability are both necessary to the advancement of scientific research, but they vary significantly in terms of their difficulty to achieve. Reproducibility, in theory, is somewhat simple to attain in data analyses–because code is inherently non-random (excepting applications involving random number generation) and data remain consistent, variability is highly restricted. The achievement of replicability, on the other hand, is a much more complex challenge, involving significantly more variablility and requiring high quality data, effective study design, and incredibly robust hypotheses.

1.2   The Reproducibility Crisis

Despite the relative simplicity of achieving reproducibility, a significant proportion of the work produced in the scientific community fails to meet reproducibility standards. 52% of respondents in a 2016 Nature survey believed that science was going through a "crisis" of reproducibility. Additionally, the vast majority of researchers across all fields studied reported having been unable to reproduce another researcher's results, while approximately half reported having been unable to reproduce their own (Baker (2016)). Other studies paint an even bleaker picture: a 2015 study found that over 50% of studies psychology failed reproducibility tests and research from 2012 found that figure closer to 90% in the field of cancer biology (Baker (2015), Begley & Ellis (2012)).

In the past several years, this "crisis" of reproducibility has risen toward the forefront of scientific discussion. Without reproducibility, the scientific community cannot properly verify study results.This makes it difficult to identify which information should be believed and which should not and increases the likelihood that studies sharing misleading information will be dispersed. The rise of data-driven technologies, alongside our newly founded ability to instantly share knowledge worldwide, has made

reproducibility increasingly critical to the advancement of scientific understanding, necessitating the development of solutions for addressing the issue.

Academics have recognized this, and publications on the topic appear to have increased siginficantly in the last several years (Eisner (2018); Fidler & Wilcox (2018); Gosselin (2020); McArthur (2019); Wallach, Boyack, & Ioannidis (2018)).

1.3 The Components of Reproducible Research

In order to see why there is an issue with reproducibility and gain a sense of how to solve it, it is important to first understand the components of reproducibility. Essentially, to answer, "What parts does researcher need to include, or what steps do they need to take, to be able to declare their work reproducible?"

Publications attempting to answer this can be found in all areas of scientific research. However, as Goodman et al. (2016) argue, the language and conceptual framework used to describe research reproducibility varies significantly across the sciences, and there are no clear standards on reproducibility agreed upon by the scientific community as a whole.

At a minimum, according to Goodman et al. (2016), achieving reproducibility requires the sharing of data (raw or processed), relevant metadata, code, and related software. However, according to other authors, the full achievement of reproducibility may require additional components.

Kitzes, Turek, & Deniz (2017) present a collection of case studies on reproducibility practices from across the data-intensive sciences, illustrating a variety of recommendations and techniques for achieving reproducibility. Although their work does not come to a consensus on the exact standards of reproducibility that should be followed, several common trends and principles emerge from their case studies that extend beyond the minimum recommendations of Goodman et al. (2016):

1) use clear separation, labeling, and documentation in provided code,
2) automate processes when possible, and
3) design the data analysis workflow as a sequence of small steps glued together, with outputs from one step serving as inputs into the next. This is a common suggestion within the computing community, originating as part of the Unix philosophy (Gancarz (2003)).

Cooper et al. (2017) focus on data analysis completed in `R` and identify a similar list of important reproducibility components, reinforcing the need for clearly labeled, well-documented, and well-separated files. In addition, they recommend publishing a list of software dependencies and using version control to track project changes over time.

Broman (2019) reiterates the need for clear naming and file separation while sharing several additional suggestions: keep the project contained in one directory, use relative paths when accessing the file system, and include a `README` file describing the project.

Wilson et al. (2017) argue that to follow good practice, an analysis must be located in a well-organized directory, be supplemented by a file containing information about the project (i.e., a README), and a provide an explicit list of dependencies.

Wilson et al. (2014) emphasize communication, recommending that code be human-readable and consistently styled for ease of understanding once shared.

The reproducibility recommendations from R OpenSci, a non-profit initiative founded in 2011 to make scientific data retrieval reproducible, share similar principles to those discussed previously. They focus on a need for a well-developed file system, with no extraneous files and clear labeling. They also reiterate the need to note dependencies and use automation when possible, while making clear a suggestion not present in the previously-discussed literature: the need to use seeds, which allow for the saving and restoring of the random number generator state, when running code involving randomness (Martinez et al. (2018)).

Although these recommendations differ from one another, when considered in combination they provide a well-rounded picture of the components important to research reproducibility across the scientific community:

1. The basic project components are made accessible to the public:

   - Data (raw and/or processed)
   - Metadata
   - Code
   - Related Software

2. The file structure of project is well-organized:

   - Separate folders for different file types.
   - No extraneous files.
   - Minimal clutter.

3. The project is documented well:

   - Files are clearly named, preferably in a way where the order in which they should be run is clear.
   - A README is present.
   - Code contains comments.
   - Software dependencies are noted.

4. File paths used in code are not system- or user-dependent:

   - No absolute paths.
   - No paths leading to locations outside of a project's directory.
   - Only relative paths, pointing to locations within a project's directory, are permitted.

5. Randomness is accounted for:

- If randomness is used in code, a seed must also be set.

6. Code is readable and consistently styled:

- Though not mentioned in the sources described previously, it is also important that code be written in a coherent style. This is because code that conforms to a style guide or is written in a consistent dialect is easier to read, simplifying the process of following a researcher's work from beginning to end (Hermans & Aldewereld (2017)).

1.4 Current Attempts to Address Reproducibility in Scientific Publishing

In an attempt to increase reproducibility, leaders from academic journals around the world have taken steps to create new standards and requirements for submitted articles. These standards attempt to address the components of reproducibility listed previously, requesting that authors provide certain materials necessary for reproducibing their work when they submit an article. However, these standards are highly inconsistent, varying significantly both across and within disciplines, and many only cover one or two of the six primary components, if any at all.

To illustrate this point, we will consider several case studies from journals publishing research on a variety of scientific fields.

1.4.1 Case Studies Across The Sciences

The journal whose requirements appear to align most closely with those components defined previously in Section 3 is the *American Journal of Political Science* (AJPS). In 2012, the AJPS became the first political science journal to require authors to make their data openly accessible online, and the publication has instituted stricter requirements since. AJPS now requires that authors submit the following alongside their papers (American Journal of Political Science (2016)).

- The dataset analyzed in the paper and information about its source. If the dataset has been processed, instructions for manipulating the raw data to achieve the final data must also be shared.
- Detailed, clear code necessary for reproducing all of the tables and figures in the paper.
- Documentation, including a README and codebook.
- Information about the software used to conduct the analysis, including the specific versions and packages used.

These standards are quite thorough and contain mandates for the inclusion of the vast majority of components necessary for complete reproducibility. Most journals, however, do not come close to meeting such high standards in their reproducibility statements.

For example, in the biomedical sciences, a group of editors reproesenting over 30 major journals met in 2014 to address reproducibility in their field, coming to a consensus on a set of principles they wanted to uphold (National Institutes of Health (2014)). Listed below are those relating specifically to the use of data and statistical methods:

1) Journals in the biomedical sciences should have a mechanism to check the statistical accuracy of submissions.

2) Journals should have no (or generous) limit on methods section length.

3) Journals should use a checklist to ensure the reporting of key information, including:

  - The article meets nomenclature/reporting standards of the biomedical field.
  - Investigators report how often each experiment was performed and whether results were substantiated by repetition under a range of conditions.
  - Statistics must be fully reported in the paper (including test used, value of N, definition of center, dispersion and precision measures).
  - Authors must state whether samples were randomized and how.
  - Authors must state whether the experiment was blinded.
  - Authors must clearly state the criteria used for exclusion of any data or subjects and must include all results, even those that do not support the main findings.

4) All datasets used in analysis must be made available on request and should be hosted on public repositories when possible. If not possible, data values should be presented in the paper or supplementary information.

5) Software sharing should be encouraged. At the minimum, authors should provide a statement describing if software is available and how to obtain it.

Even though these principles seem well-developed on the surface, they fail to meet even the basic requirements defined by Goodman et al. (2016) previously. Several of the principles are purely recommendations; there is no requirement that code be shared, nor metadata. Additionally, software requirements are quite loose, requiring no information about dependencies or software version.

We see a similar issue even in journals designed specifically for the purpose of improving scientific reproducibility. *Experimental Results*, a publication created by Cambridge University Press to address some of the reproducibility and open access issues in academia, also falls short of meeting high standards. The journal, which showcases articles from a variety of scientific decisions disciplines, states in their transparency and openness policy:

> Whenever possible authors should make evidence and resources that under-pin published findings, such as data, code, and other materials, available to readers without undue barriers to access.

The inclusion of code and data are only recommended and no definition of what "other materials" may mean is provided. No components of reproducibility extending beyond those required at a minimum are even considered (Cambridge University Press (2020)).

The *American Economic Review*, the first of the top economics journals to require the inclusion of data alongside publications, has stronger guidelines than several of those mentioned previously, though not as strong as the *American Journal of Political Science*. Their Data and Code Availability Policy states the following (American Economic Association (2020)):

> It is the policy of the American Economic Association to publish papers only if the data and code used in the analysis are clearly and precisely documented, and access to the data and code is clearly and precisely documented and is non-exclusive to the authors.

These requirements are quite strict, prohibiting exceptions for papers using data or code not available to the public in the way that many other journals claiming to promote reproducibility do.

1.4.2 Case Studies In The Statistical And Data Sciences

When considering reproducibility policy, the field of Statistical and Data Sciences performs relatively well. The majority of highly ranked journals in the field contain statements on reproducibility. Some of these are quite robust, surpassing the requirements of many of the other journals discussed previously, while others are lacking.

The *Journal of the American Statistical Association* stands out as having relatively robust requirements. The publication's guideliens require that data be made publicly available at the time of publication except for reasons of security or confidentiality. It is strongly recommended that code be deposited in open repositories. If data is used in a process form, the provided code should include the necessary cleaning/preparation steps. Data must be in an easily understood form and a data dictionary should be included. Code should also be in a form that can be used and understood by others, including consistent and readable syntax and comments. Workflows involving more than one script should also contain a master script, Makefile, or other mechanism that makes it clear what each component does, in what order to run them, and what the inputs and outputs to each area (American Statistical Association (2020)).

The *Journal of Statistical Software* also has strong guidelines, though less thorough. Authors must provide *commented* source code for their software; all figures, tables, and output must be exactly reproducible on at least one platform, and random number generation must be controlled; and replication materials (typically in the form of a script) must be provided (Journal of Statistical Software (2020)).

The expectations of the *Journal of Computational and Graphical Statistics* are notably weaker, requiring only that authors "submit code and datasets as online supplements to the manuscript," with exceptions for security or confidentiality, but providing no further detail (Journal of Computational and Graphical Statistics (2020)).

The *R Journal* has the same requirements, but with no exceptions on the data provision policy, stating that authors should "not use such datasets as examples" (R Journal Editors (2020)).

Perhaps the weakest reproducibility policies come from *The American Statistician* and the *Annals of Statistics*. The former appears to have no requirements, stating only that it "strongly encourages authors to submit datasets, code, other programs, and/or appendices that are directly relevant to their submitted articles," while the latter appears to have no statement on reproducibility at all (The American Statistician (2020)).

Journal Name	Code Sharing Required?	Data Sharing Required?	Other Components Required?
Journal of the American Statistical Association	+ ◆	+	+
Journal of Statistical Software	+	+	+
Journal of Computational and Graphical Statistics	+ ◆	+ ◆	🚫
The R Journal	+	+	🚫
The American Statistician	🚫 ★	🚫 ★	🚫
The Annals of Statistics	🚫	🚫	🚫

★ Component recommended, but not required.
◆ Component required, but exceptions granted.

Figure 1.2: Reprodubility Policies of Top Statistical and Data Sciences Journals

1.4.3   The Bigger Picture

The journals mentioned here are just some of the many academic publishers with reproducibility policies. While they provide a sense of the specific wording and requirements of some policies, they do not necessarily serve as a representative sample of all academic publishing. It is important to also consider the bigger picture, exploring the state of reproducibility policy in academic publishing as a whole.

Given the scale of the academic publishing network and the sheer number of journals around the world, this is not necessarily an easy task.

In order to simplify this process, academics at the Center for Open Science (COS) attempted to create a metric, called the TOP Factor. The TOP Factor reports the steps that a journal is taking to implement open science practices. It has been calculated for a wide variety of journals, though the COS is still far from scoring all of the publications that are currently available.

The TOP Factor is calculated as follows. Publications are scored on a variety of categories associated with open science and reproducibility. For each category, they receive a score between 0 (poor) and 3 (excellent) based on the degree to which they emphasize each category in their submission/publication policies. A journal's final score, which can range from 0 to 30, is the sum of the individual scores in each of the categories.

	Not Implemented	Level I	Level II	Level III
Citation standards	Journal encourages citation of data, code and materials or says nothing.	Journal describes citation of data in guidelines to authors with clear rules and examples	Article provides appropriate citation for data and materials used consistent with journal's author guidelines	Article is not published until providing appropriate citation for data and materials following journal's author guidelines
Data transparency	Journal encourages data sharing or says nothing	Article states whether data are available and, if so, where to access them	Data must be posted to a trusted repository. Exceptions must be identified at article submission	Data must be posted to a trusted repository, and reported analyses will be reproduced independently prior to publication
Code transparency	Journal encourages code sharing or says nothing	Article states whether code is available and, if so, where to access it	Code must be posted to a trusted repository. Exceptions must be identified at article submission	Code must be posted to a trusted repository, and reported analyses will be reproduced independently prior to publication
Research materials transparency	Journal encourages materials sharing or says nothing	Article states whether materials are available and, if so, where to access them	Materials must be posted to a trusted repository. Exceptions must be identified at article submission	Materials must be posted to a trusted repository, and reported analyses will be reproduced independently prior to publication

Design and analysis transparency	Journal encourages design and analysis transparency or says nothing	Journal articulates design transparency standards	Journal requires adherence to design transparency standards for review and publication	Journal requires and enforces adherence to design transparency standards for review and publication
Study preregistration	Journal says nothing	Article states whether preregistration of study exists and, if so, where to access it	Article states whether preregistration of study exists and, if so, allows journal access during peer review for verification	Journal requires preregistration of studies and provides link and badge in article to meeting requirements
Analysis plan preregistration	Journal says nothing	Article states whether preregistration of study exists and, if so, where to access it	Article states whether preregistration with analysis plan exists and, if so, allows journal access during peer review for verification	Journal requires preregistration of studies with analysis plans and provides link and badge in article to meeting requirements
Replication	Journal discourages submission of replication studies or says nothing	Journal encourages submission of replication studies	Journal encourages submission of replication studies and conducts results blind review	Journal uses registered reports as a submission option for replication studies with peer review prior to observing the study outcomes

Figure 1.3: The TOP Factor Rubric

When looking at the overall distribution of TOP Factor scores, we see a relatively grim picture: Around 50% of journals score as low as 0-5 overall, while only just over 5% score more than 15, just half of the maximum possible score. Over 40 journals failed to score a single point (Woolston (2020)).

Although it is clear that some journals have relatively strong reproducibility and openness policies, that is clearly not the norm. And many that do appear to have policies lacking in robustness, including exceptions for data privacy and security concerns or phrasing guidelines as recommendations rather than reqquirements. The field of data science stands out among the rest, with the majority of top journals

having relatively robust policies.

1.4.4   Assessing the Success of Academic Reproducibility Policies

We have seen that, although not necessarily the standard, some journals across the sciences have enacted reproducibility policies. The simple implementation of a policy, however, does not ensure that its goals will be achieved. Reproducibility can only be addressed when both authors *and* journal reviewers actively implement publishing standards in practice. Without participation and dedication from all involved, reproducibility guidelines serve more as a theoretical goal than a practical achievement.

It is important to ask, then, whether academic reproducibility standards *actually* result in a greater number of reproducible publications.

Let us consider the case of the journal *Science. Science* instituted a reproducibility policy in 2011 and has maintained it ever since. In its original form, their policy stated the following:

> All data necessary to understand, assess, and extend the conclusions of the manuscript must be available to any reader of Science. All computer codes involved in the creation or analysis of data must also be available to any reader of Science. After publication, all reasonable requests for data and materials must be fulfilled. Any restrictions on the availability of data, codes, or materials. . . must be disclosed to the editors upon submission. . .

This policy is similar to many of the others considered previously, requiring the publishing of code and data with exceptions permitted when necessary.

Stodden, Seiler, & Ma (2018a) tested the efficacy of this policy in practice, emailing corresponding authors of 204 articles published in the year after *Science* first implmented its policy to request the data and code associated with their articles. The researchers only received (at least some of) the requested material from 36% of authors. This low rates were due to several factors:

- 26% of authors did not respond to email contact.
- 11% of authors were unwilling to provide the data or code without further information regarding the researchers' intentions.
- 11% asked the researchers to contact someone else and that person did not respond.
- 7% refused to share data and/or code.
- 3% directed the researchers back to their paper's supplmental information section.
- 3% of authors made a promise to follow up and then did not follow through.
- 3% of emails bounced.
- 2% gave reasons why they could not share for ethical reasons, size limitations, or some other reason.

Of the 56 papers they deemed likely reproducible, the authors randomly selected 22 and were able to replicate the results for all but 1, which failed due to its reliance on software that was no longer available.

Hardwicke et al. (2018) conducted a study on the journal *Cognition*, where researchers compared the reproducibility of published work both before and after the journal instituted an open data policy, which required that authors make relevant research data publicly available prior to publication of an article.

The researchers found a considerable increase in the proportion of data available statements (in constrast to 'data not available' statements, which could be present due to privacy or security concerns) since the implementation of the policy. Pre-open data policy, only 25% of articles had data available, while that number was a much higher 78% after the policy was put in place.

While the institution of an open data policy appears to have been associated with a significant increase in the percentage of studies with data available, further research indicates that the policy was perhaps not as effective as intended. Many of the datasets were usable in theory, but not in practice. Only 62% of the articles with data available statements had truly reusable datasets–in this case, meaning that the data were accessible, complete, and understandable. Though this is an increase from the pre-policy period, which saw 49% of articles with data availability statements as reusable in practice, it is still far from ideal.

Combining these two data points indicates that *less than half* of articles published after the open data policy was instituted actually contained truly usable data.

In this small sample of cases, we see that purely having a reproducibility statement does not necessarily mean that all, or even a majority, of published work will truly be reproducible.

1.5 Limitations on Achieving Reproducibility in Scientific Publishing

There are several reasons for this apparent divide between journal reproducibility standards and the true proportion of submitted articles that are truly reproducible. Some of these are challenges faced by the article authors, while others are faced by the journal editors.

1.5.1 Challenges for Authors

Stodden, Seiler, & Ma (2018b) conducted a survey asking over 7,700 researchers about one of the key characteristics of reproducibility – open data – and gathered information about the reasons why authors found difficulties in making their data available to the public

The main challenges listed by respondents were as follows:

- 46% identified "Organizing data in a presentable and useful way" to be difficult.
- 37% had been "Unsure about copyright and licensing."

- 33% had problems with "Not knowing which repository to use."
- 26% cited a "Lack of time to deposit data."
- 19% found the "Costs of sharing data" to be high.

The relative frequency of these issues varied across several characteristics, including author seniority, subject area, and geographical location, though authors in all categories faced some issues.

Beyond technical challenges, other reasons may lead authors to not place their focus on reproducibility. For example, some researchers might fear damage to their reputation if a reproduction attempt fails after they have provided the necessary materials (Lupia & Elman (2014)).

Given the relatively high frequency of concern over achieving reproducibility, it follows that researchers will not make the necessary effort to do so if journal guidelines provide a way out. Policies that *recommend* the inclusion of data or that allow exceptions to open data for certain reasons are likely to be associated with a lower proportion of reproducible articles than those that make open data mandatory.

1.5.2   Challenges for Journals

In addition to the challenges faced on the part of the authors, journal reviewers face their own difficulties in ensuring reproducibility.

In order to make sure that all submitted articles comply with reproducibility guidelines, reviewers must go through them one by one and reproduce all of the results by hand using the provided materials.

This is an incredibly intensive process, as we will see in the example of the *American Journal for Political Science* (AJPS), whose reproducibility policy was discussed prevously in Chapter 1.4.1.

Jacoby, Lafferty-Hess, & Christian (2017) describe the AJPS process in detail:

Acceptance of an article for publication in the AJPS is contingent on successful reproducibility of any empirical analyses reported in the article.

After an article is submitted, staff from a third party vendor hired by AJPS go through the provided materials to ensure that they can be preserved, understood, and used by others. They then run all of the analyses in the article using the code, instructions, and data provided by the authors and compare their results to the submitted articles. Authors are then given an opportunity to resolve any issues that come up. This process is repeated until reproducibility is ensured.

Although providing a significant benefit to the scientific community, this thorough process is associated with high costs.

The verification process slows down the journal review process significantly, adding a median 53 days to the publication workflow, as many submitted articles require one or more rounds of resubmission (the average number of resubmissions is 1.7). It is also quite labor intensive, taking an average of 8 person-hours per manuscript to reproduce the analyses and prepare the materials for public release and adding significant monetary cost to AJPS.

Journals are often reluctant to take on such an intensive task due to the drastically

increased burden it places on reviewers and on the publication's financial resources. This is particularly true given that the number of submitted articles per year has been increasing over time (Leopold (2015)). Every additional submission increases the burden of achieving reproducibility, and with a large enough volume, the challenge can quickly become seemingly impossible to manage reasonably.

As a result, journals often encourage reviewers to consider authors' compliance with data sharing policies, but do not formally require that they ensure it as a criterion for acceptance (Hrynaszkiewicz (2020)).

1.6 Attempts to Address These Limitations

The previous discussion makes clear that, although reproducibility is critically important to scientific progress and academic journals are taking steps to encourage it, the scientific community is far from achieving the desired level of widespread reproducibility. In large part, this appears due to the challenge and complexity of actually achieving reproducibility. Those attempting to improve the reproducibility of work can face issues with concerns over legality of sharing data, large commitments of time or money, challenges finding a good repository, and organizing all of the many components of their work in an understandable way, among other things.

Additionally, science faces the additional challenge that many publishers do not emphasize reproducibility at all, providing many opportunities for all authors except those personally dedicated to producing reproducible work to leave reproducibility by the wayside. Many journals have no reproducibility requirements, and those that do often do not take the necessary steps to ensure that they are actually met.

These issues, however, are not impossible to overcome. Proponents of reproducibility have taken action to help address them, both through education on reproducibility and through software that helps simplify the process of achieving it.

1.6.1 Through Education

One way to address the reproducibility crisis is to educate data analysts on the topic so that they are aware of both the concept of reproducibility and how to achieve it in their own work. A natural place to focus this education is early on in the data science training pipeline as part of introductory or early-intermediate courses in undergraduate and graduate data science porgrams (Horton, Baumer, & Wickham (2014)). This sort of educational integration has a variety of benefits:

- Bringing reproducibility into the discussion early on gives students the tools to add knowledge to their field in the best way possible before they actually conduct any substantive analysis on their own (Janz (2016)). This produces many long run benefits, helping lessen the burden on promoting reproducibility placed on journals and increasing the number (and percentage) of researchers doing and promoting reproducible work.

- If covered in detail as part of the data science curriculum, reproducibility will

eventually come easily to students. If learned independently, without effective tools, reproducibility can challenging and even disheartening to try to understand and succeed at. Practicing in the classroom gives students the ability to fail without damaging their reputation, giving a great opportunity to truly learn and understand the concepts so that they feel capable of handling them when they begin their own research.

- The application of grading to the topic provides an incentive for students to pay attention, learn, and absorb the information. This same incentive does not exist when researchers attempt to learn about reproducibility independently. In that situation, internal motivation, which may be weak in some individuals, is the only factor present to help promote success.

Several educators, primarily at the graduate level, have realized the opportunity and taken steps to introduce reproducibility into their courses. According to Janz (2016), the primary way of achieving this integration is through the assignment of "replication studies" in standard methods choices. In these assignments, students are given a published study and its supporting materials and asked to reproduce the results. The most famous course of this kind is Government 2001, taught by Gary King at Harvard University. In King's course, students team up in small groups to reproduce a previous study. To help ensure that their workflow is reproducible, students are required to hand over their data and code to another student team who then tries to reproduce their work once again.

In Thomas M. Carsey's intermediate statistics course at the University of North Carolina at Chapel Hill, students must reproduce the findings of a study by re-collecting the data from the original sources, then must extend the study by building on the analysis.

Christopher Fariss of Penn State University asks his students to replicate a research paper published in the last five years, noting that students must describe the article and the ease in which the results replicate.

The University of California at Berkeley has a similar course to Gary King's Harvard course, where students each take a different piece of an existing study to work on reproducing and have to ensure that their piece fits with the piece of the next student (Hillenbrand (2014)).

At the undergraduate level, rather than assign replication studies the way many graduate schools tend to do, Smith College and Duke University have both integrated reproducibility into their introductory courses through the requirement that assignments be completed in the `RMarkdown` code + narration format (B. Baumer, Cetinkaya-Rundel, Bray, Loi, & Horton (2014)).

Another way to provide education on reproducibility is through the creation of workshops that focus solely on the topic, rather than through integration as just one part of a class (@Janz (2016)).

For example, the University of Cambridge conducts a Replication Workshop, where graduate students are asked replicate a paper in their field over eight weekly sessions. When students encounter challenges, such as authors not responding to queries for

data or steps of the analysis being poorly defined and explained, they gain a first hand understanding of the consequences of poor transparency.

Workshops such as these are typically optional and not included as part of the primary curriculum, however, so while they may cover the topic of reproducibility in more detail than traditional courses, they often reach fewer students.

In spite of all of the advantages that these educational tools provide, "reproducibility training and assessment in data science education is largely neglected, especially among undergraduates and Master's students in professional schools..., probably because the students are usually considered to be non-research oriented" (Yu & Hu (2019)). While some examples of reproducibility education exist, they are certainly not commonplace. However, given the increased discussion and emphasis on reproducibility in academia over the past several years, it is likely that this will change, particularly if methods are provided to educators to make the integration of reproducibility into their courses simple and relatively unburdensome.

1.6.2   Through Software

Several researchers and members of the Statistical and Data Sciences community have taken action to develop software focused on reproducibility which removes some of the load on data analysts by automating reproducibility processes and checking whether certain components are achieved.

Much of this software has been written for users of the coding and data analysis language `R`. `R` is very popular in the data science community due to its open-source nature, accessibility, extensive developer and user base, and statsitical analysis-specific features.

Some of the existing software solutions are listed below:

`rrtools` (Marwick (2019)) addresses many of the issues discussed in Marwick, Boettiger, & Mullen (2018) by creating a basic reproducible structure based on the `R` package format for a data analysis project. In addition, it allows for isolation of the computer environment using `Docker`, provides a method to capture information about the versions of packages used in a project, contains tools for generating a README file, and provides an option for users to write tests to check that their functions operate as intended.

The `orderly` (FitzJohn et al. (2020)) package also focuses on file structure, requiring the user to declare a desired project structure (typically a step-by-step structure, where outputs from one step are inputs into the next) at the beginning and then creating the files necessary to achieve that structure. Its principal aim is to automate many of the basic steps involved in writing analyses, making it simple to:

1) Track all inputs into an analysis.
2) Store multiple versions of an analysis where it is repeated.
3) Track outputs of an analysis.
4) Create analyses that depend on the outputs of previous analyses.

When projects have a variety of components, `orderly` makes it easy to see inputs and outputs change with each re-run.

`workflowr`'s (Blischak, Carbonetto, & Stephens (2019)) functionality is based around version control and making code easily available online. It works to generate a website containing time-stamped, versioned, and documented results. In addition, it manages the session and package information of each analysis and controls random number generation.

`checkers` (Ross, DeCicco, & Randhawa (2018)) allows you to create custom checks that examine different aspects of reproducibility. It also contains some pre-built checks, such as seeing if users reference packages that are less preferred to other similar ones and ensuring that the project is under version control. `renv` (Ushey & RStudio (2020)) (formerly `packrat`) helps to make projects more isolated, portable, and reproducible. It gives every project its own private package library, makes it easy to install the packages the project depends on if it is moved to another computer.

`drake` (OpenSci (2020)) analyzes workflows, skips steps where results are up to date, utilizes optimized computing to complete the rest of the steps, and provides evidence that results match the underlying code and data.

Lastly, the `reproducible` (McIntire & Chubaty (2020)) package focuses on the concept of caching: saving information so that projects can be run faster each time they are re-completed from the start.

Figure 1.4: Popular Reproducibility Packages in R

There have also been several `Continuous integration` tools developed outside of R which can be used by those coding in almost any language. These provide more general approaches to automated checking, which can enhance reproducibility with minimal code.

For example, `wercker`—a command line tool that leverages Docker—enables users to test whether their projects will successfully compile when run on a variety of operating systems without access to the user's local hard drive (Oracle Corporation (2019)).

`GitHub Actions` is integrated into GitHub and can be configured to do similar checks on projects hosted in repositories.

`Travis CI` and `Circle CI` are popular continuous integration tools that can also be used to check `R` code.
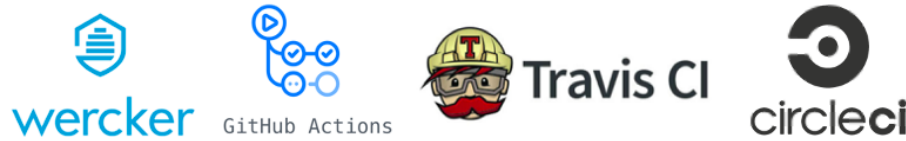
Figure 1.5: Popular Continuous Integration Tools

1.7 Understanding The Gaps In Existing Reproducibility Solutions

Although the current state of reproducibility in academia is quite poor, it is not an impossible challenge to overcome. The relative simplicity of addressing reproducibility, particularly when compared with replicability, makes it an ideal candidate for solution-building. Although significant progress on addressing reproducibility on a widespread scale is a long-term challenge, impactful forward progress–if on a smaller scale–can be achieved in the short-term.

As we have seen, software developers, data scientists, and educators around the world have realized this potential, taking steps to help address the current crisis of reproducibility. Journals have put in place guidelines for authors, statisticians have developed `R` packages that help structure projects in a reproducible format, and educators have begun integrate reproducibility exercises into their courses.

We have already explored the issues with journal policies, both for authors and reviewers, in-depth. Educational and software-based strategies attempt to address these policy issues by spreading ideas of reproducibility to more people and simplifying the process of achieving it, therefore resulting (in theory) in an improvement in the reproducibility of published scientific articles.

However, the reality is not quite so rosy. Many current educational and software-based solutions face their own challenges that limit them from achieving the desired outcome. In this section, we will consider these issues.

1.7.1 In Education

The two primary concerns about the integration of reproducibility in data science curricula revolve around time and difficulty.

As noted previously, the primary mode of teaching reproducibility is through the assignment of replication studies where students must take an existing study and go through the process of reproducing it themselves, including contacting the author for all necessary materials, rerunning code and analysis, and problem-solving when issues almost certainly come up.

In addition to the time required for the professor to collect all of the studies that students will be working on, the inclusion of such an assignment places a significant burden on educators by taking up time where they could be teaching other important material. Replication studies, if done correctly, can take weeks for students to

successfully complete. The choice to give such assignments is therefore associated with a significant opportunity cost which many professors are unwilling to take.

Additionally, both replication studies assigned in class and replication workshops outside of normal coursework require a working knowledge of how to successfully complete and understand research. This makes them inaccessible to individuals who are still in their undergraduate career and may not yet have had an opportunity to conduct research or those who are studying in non-research-focused technical programs.

In order to reach the widest variety of students possible, it is necessary to develop a new method of teaching reproducibility that is neither time consuming nor dependent on a prior understanding of the research process.

1.7.2   In Software

Previously, we considered several different types of software solutions: packages designed for users of R and continuous integration programs that can be used alongside a variety of coding languages. Although these solutions have their advantages, they also have significant drawbacks in terms of their ability to address reproducibility on a widespread scale.

Many of the packages designed for R are incredibly narrow in scope, with each effectively addressing a small component of reproducibility: file structure, modularization of code, version control, etc. They often succeed in their area of focus, but at the cost of accessibility to a wider audience. Their functions are often quite complex to use, and many steps must be completed to achieve the required reproducibility goal. This cumbersome nature means that most reproducibility packages currently available are not easily accessible to users with minimal R experience, nor particularly useful to those looking for quick and easy reproducibility checks. The significant learning curve associated with them can also detract potential users who may be interested in reproducibility but not willing to dedicate an extensive amount of time to understanding the intricacies of software operation.

Due to their generalized design, Continuous Integration tools do not face the same issues with narrowness or complexity that R packages struggle with. However, this generalizability provides its own additional challenge. Since Continuous Integration tools are designed to be accessible to a wide variety of users with different coding preferences, they are not particularly customizable and lack the ability to address features specific to certain programming languages.

Neither of these different software solutions appear to adequately address the challenge of reproducibility.

1.7.3   What We Need Moving Forward

While a variety of attempts to address reproducibility have been made, they all face their own set of challenges. Most focus on only one area of reproducibility, are too time consuming and burdensome to attempt, or require an extensive amount of background knowledge.

In order to truly improve scientific reproducibility, a better solution is needed.

The optimal solution should combine the positive aspects of the previously discussed educational and software methods, while remaining simple and easy to use. It should also have a variety of potential applications—journal reviewers should benefit from it, as should authors, as should educators and students and even those outside of academia.

The full list of necessary features for an effective reproducibility tool are provided below:

1) Be simple, with a small library of functions/tools that are straightforward to use.
2) Be accessible to a variety of users, with a relatively small learning curve.
3) Be able to address a wide variety of aspects of reproducibility, rather than just one or two key issues.
4) Have features specific to a particular coding language that can address that language's unique challenges.
5) Be customizable, allowing users to choose for themselves which aspects of reproducibility they want to focus on.
6) Be educational, teaching those that use it about why their projects are not reproducible and how to correct that in the future.
7) Be applicable to a wide variety of domains.

While this seems like a lot to ask, such a task is not impossible. In the next chapter, we will consider one potential solution: `fertile`, an R package focused on reproducibility that I have worked to develop.

Chapter 2

`fertile`: My Contribution To Addressing Reproducibility

2.1 `fertile`, An R Package Creating Optimal Conditions For Reproducibility

What if it were possible to address the existing issues with both educational and software reproducibility solutions simultaneously?

That is where my work comes in. In an attempt to produce meaningful change in the field of reproducibility, I have been developing `fertile`, a software package designed for `R` which helps users create optimal conditions for achieving reproducibility in their projects.

Figure 2.1: fertile's Package Logo

`fertile` attempts to address the gaps in existing reproducbility solutions by combining software and education in one product. The package provides a set of simple, easy-to-learn tools that, rather than focus intensely on a specific area like other software programs, provide some information about all six major components of reproducibility. It is also designed to be incredibly flexible, offering benefits to users at any stage in the data analysis workflow and providing users with the option to

select which aspects of reproducibility they want to focus on.

`fertile` also contains several `R`-specific features, which address certain aspects of reproducibility that can be missed by external project development tools. It is designed primarily to be used on data analyses organized as `R` Projects (i.e. directories containing an `.Rproj` file) and contains several associated features to ensure that the project structure meets the standards discussed in the `R` community.

In addition, `fertile` is designed to be educational, teaching its users about the components of reproducibility and how to achieve them in their work. The package provides users with detailed reports on the aspects of reproducibility where their projects fell short, identifying the root causes and, in many cases, providing a recommended solution.

`fertile` is structured in such a way as to be understandable and operable to individuals of any skill level, from students in their first undergraduate data science course to experienced PhD statisticians. The majority of its tools can be accessed in only a handful of functions with minimal required arguments. This simplicity makes the process of achieving and learning about reproducibility accessible to a wide audience in a way that complex software programs or graduate courses requiring an advanced knowledge of research methods do not.

Reproducibility is significantly easier to achieve when all of the tools necessary to do so are located in one place. `fertile` provides this optimal all-inclusive structure, addressing all six of the primary components of reproducibility discussed in Chapter 1. We will consider `fertile`'s treatment of each of these components in turn, exploring its analysis of the sample R project shown below, titled `project_miceps`:
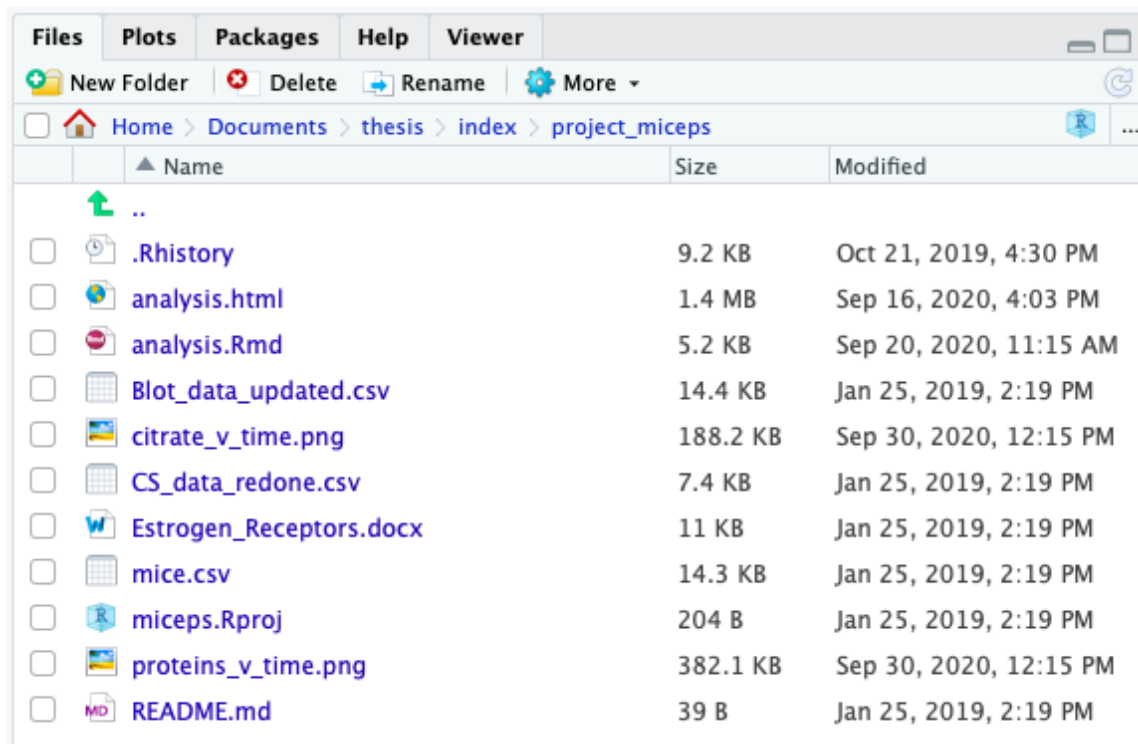
Figure 2.2: A Sample R Project

2.1.1   Component 1: Accessibile Project Files

```r
library(fertile)
```

fertile takes several steps to help users ensure that all of files necessary to run an analysis are provided in the project folder.

File Overview

One of the fastest ways to gain an overview of the existing files (including data, code, and metadata) is with the `proj_analyze_files()` function. It lists all of the files in the project, along with their size, type, and relative path within the directory. This can help users quickly produce an overview of how many code, data, and auxilliary (image or text) files they have.

```r
proj_analyze_files("project_miceps")
```

```
# A tibble: 11 x 5
   file                    size ext   mime                         path_rel
   <fs::path>          <fs::by> <chr> <chr>                        <fs::path>
 1 project_miceps/Blot~  14.43K csv   text/csv                     Blot_data_upd~
 2 project_miceps/CS_d~   7.39K csv   text/csv                     CS_data_redon~
 3 project_miceps/Estr~  10.97K docx  application/vnd.openxmlfo~    Estrogen_Rece~
 4 project_miceps/READ~      39 md    text/markdown                README.md
 5 project_miceps/anal~   5.21K Rmd   text/x-markdown              analysis.Rmd
 6 project_miceps/anal~   1.41M html  text/html                    analysis.html
 7 project_miceps/citr~ 187.12K png   image/png                    citrate_v_tim~
 8 project_miceps/mice~  14.33K csv   text/csv                     mice.csv
 9 project_miceps/mice~     204 Rproj text/rstudio                 miceps.Rproj
10 project_miceps/prot~ 378.75K png   image/png                    proteins_v_ti~
11 project_miceps/soft~   5.34K txt   text/plain                   software-vers~
```

Users can also check for the existence of a README description file with `has_readme()`:

```r
has_readme("project_miceps")
```

```
v Checking for README file(s) at root level
```

Testing For Self-Containment

In order to truly check that a project is self contained, however, is is necessary to remove it from the environment it is located, ensuring that it can still run when cut off from the main file system.

The `sandbox()` function is designed to help facilitate this. `sandbox()` allows the user to make a copy of their project in a temporary directory that is isolated from the file system.

```
fs::dir_ls('project_miceps') %>% head(3)
```

```
project_miceps/Blot_data_updated.csv    project_miceps/CS_data_redone.csv
project_miceps/Estrogen_Receptors.docx
```

```
temp_dir <- sandbox('project_miceps')
temp_dir
```

```
/var/folders/v6/f62qz88s0sd5n3yqw9d8sb300000gn/T/RtmpMxMwKV/project_miceps
```

```
fs::dir_ls(temp_dir) %>% head(3)
```

```
/var/folders/v6/f62qz88s0sd5n3yqw9d8sb300000gn/T/RtmpMxMwKV/project_miceps/Blot_data_updat
/var/folders/v6/f62qz88s0sd5n3yqw9d8sb300000gn/T/RtmpMxMwKV/project_miceps/CS_data_redone.
/var/folders/v6/f62qz88s0sd5n3yqw9d8sb300000gn/T/RtmpMxMwKV/project_miceps/Estrogen_Recept
```

Once a directory is sandboxed, users can run the `proj_render()` function, which checks all `.R` and `.Rmd` code files in the directory to ensure that they can compile properly, to ensure that their project is self-contained.

```
proj_render(temp_dir)
```

```
-- Rendering R scripts... ----------------------------- fertile 0.0.0.9029 --
```

If `proj_render()` executes without error, this indicates that all the necessary files are present. Users can also confirm this by checking the time stamp of the last successful render, captured in the `render_log_report()` function. If the time stamp matches the current time, the project successfully compiled.

```
slice_tail(render_log_report(temp_dir))
```

```
Reading from /var/folders/v6/f62qz88s0sd5n3yqw9d8sb300000gn/T/RtmpMxMwKV/project_miceps/.f
```

```
# A tibble: 1 x 4
  path          path_abs func       timestamp
  <chr>         <chr>    <chr>       <dttm>
1 LAST RENDERED <NA>     proj_render 2020-11-02 23:54:54
```

2.1.2 Component 2: Organized Project Structure

`fertile` provides a wide variety of features for managing the file system of a project. Nine of the package's fifteen primary reproducibility checks relate to file structure.

Clear Project Root

Two of these are focused on the `R project` aspect of the file system. `has_proj_root()` ensures that there is a single `.Rproj` file indicating a clear root directory for the project, while `has_no_nested_proj_root()` ensures that there are no sub-projects within. The recognition of a clear root directory is necessary to allow for file structure analysis and project restructuring as it provides a baseline directory to define relative file paths from.

No File Clutter

Six of the major checks, whose names begin with `has_tidy_` focus on file clutter, addressing one of the components of a clean file structure. They check to make sure that no audio/video, image, source, raw data, .rda, or .R files are found in the root directory of the project.

   The last check that focuses on the file system is `has_only_used_files()`. One of the more complicated checks in `fertile`, this function checks to make sure that there are no extraneous files serving no purpose (that are not "used") included alongside the analysis. For this function to work properly, a clear definition of which files are considered as "used" is needed. In `fertile`, that definition includes the following:

- `.R`,`.Rmd`, and `.Rproj` files
- README files
- Data/text/image files whose file path is referenced, either as an input or output, in any of the `.R` or `.Rmd` files
- Outputs created by knitting any `.Rmd` files (`.html`, `.pdf`, or `.docx` files with the same name as the `.Rmd`)
- Files created by `fertile`. In an effort to capture information about dependencies, `fertile` creates a text file capturing the session information when a project is run and provides an option to generate an `.R` script that can install all of the packages referenced in a project's code. These files are considered "used."

```r
# List the files in the directory
fs::dir_ls("project_miceps")
```

```
project_miceps/Blot_data_updated.csv    project_miceps/CS_data_redone.csv
project_miceps/Estrogen_Receptors.docx  project_miceps/README.md
project_miceps/analysis.Rmd             project_miceps/analysis.html
project_miceps/citrate_v_time.png       project_miceps/mice.csv
project_miceps/miceps.Rproj             project_miceps/proteins_v_time.png
project_miceps/software-versions.txt
```

```r
# Check to see which are "used"
has_only_used_files("project_miceps")
```

```
Joining, by = "path_abs"
```

```
* Checking to see if all files in directory are used in code

  Problem: You have files in your project directory which are not being used.

  Solution: Use or delete files.

  See for help: ?fs::file_delete
```

```
# A tibble: 2 x 1
  path_abs
  <chr>
1 /Users/audreybertin/Documents/thesis/index/project_miceps/Estrogen_Receptors.~
2 /Users/audreybertin/Documents/thesis/index/project_miceps/mice.csv
```

Reorganizing File Structure

There are also several functions help with reshaping the entire project structure to a more reproducibile format. For example, `proj_suggest_moves()` provides suggestions for how to reorganize the files into folders.

```
files <- proj_analyze_files("project_miceps")

proj_suggest_moves(files)
```

```
# A tibble: 9 x 8
  file             size ext   mime      path_rel   dir_rel    path_new    cmd
  <fs::path>   <fs::by> <chr> <chr>     <fs::path> <fs::path> <fs::path>  <chr>
1 project_mic~   14.43K csv   text/csv  Blot_data~ data-raw   project_mi~ file_m~
2 project_mic~    7.39K csv   text/csv  CS_data_r~ data-raw   project_mi~ file_m~
3 project_mic~   10.97K docx  applica~  Estrogen_~ inst/other project_mi~ file_m~
4 project_mic~    5.21K Rmd   text/x-~  analysis.~ vignettes  project_mi~ file_m~
5 project_mic~    1.41M html  text/ht~  analysis.~ inst/text  project_mi~ file_m~
6 project_mic~  187.12K png   image/p~  citrate_v~ inst/image project_mi~ file_m~
7 project_mic~   14.33K csv   text/csv  mice.csv   data-raw   project_mi~ file_m~
8 project_mic~  378.75K png   image/p~  proteins_~ inst/image project_mi~ file_m~
9 project_mic~    5.34K txt   text/pl~  software-~ inst/text  project_mi~ file_m~
```

These suggestions are based on achieving the optimal file structure design for reproducibility, argued by Marwick et al. (2018) to be that of an R package (R-Core-Team (2020), Wickham (2015)), with an R folder, as well as `data`, `data-raw`, `inst`, and `vignettes`. Such a structure keeps all of the files separated and in clearly labeled directories where they are easy to find. Additionally, the extension of the R package structure to data analysis projects promotes standardization of file structure within the R community.

Users can execute these suggestions individually, using the code snippets provided next to each file name when running `proj_suggest_moves()`, but they can also

run them all together. `proj_move_files()` executes all of the suggestions from `proj_suggest_moves()` at once, building an R package style structure and sorting files into the correct folders by type.

Combined together, all of these functions make it simple to ensure that projects fall under a standard, simple, and reproducible structure with minimal clutter.

2.1.3   Component 3: Documentation

High quality documentation, including the presence of a README file, comments explaining the code, a list of software packages an analysis is dependent on, and a method in which to understand which order to run code files in, is incredibly important to reproducibility. Without written guidance, individuals looking to reproduce results may not understand how to take all of the project components and combine them in the right way to produce the desired outcome. Additionally, code used without the proper dependencies and software versions, even if it is perfectly functional and correctly written, will often result in errors when compiled.

`fertile` contains a variety of functions to ensure that projects are well documented.

README

One important component of documentation is the inclusion of a README file. A README is a text file that introduces and explains a project. Commonly, a project README contains background information necessary to understand a project. Some of the components that might be contained within are:

1. Background on the purpose of the project and the questions of interest.
2. Background on where the data used in the project came from and what they contain.
3. Information about installing and setting up the software necessary to run it.
4. A list of included files.
5. A description of how to run the project—for example, a summary of the order in which the included files should be run.
6. Contact information for the author of the project.

`fertile` ensures that a README is included alongside the user's project with the `has_readme()` function:

```
has_readme("project_miceps")
```

v Checking for README file(s) at root level

Clear File Order

Another important component of documentation is that it must be clear in which order provided files should be run. While this information can be included as part of the README, it is sometimes provided via other methods—for example, through

a `makefile`/`drakefile` (make-style file generated by the `Drake` package in `R`) or through a standardized naming or numbering convention of files.

`has_clear_build_chain()` checks for these non-README methods of ensuring that file ordering is clear. `project_miceps`, since it only contains one code file, passes this check easily:

```
has_clear_build_chain("project_miceps")
```

v Checking for clear build chain

Software Dependencies

There are also several features focused on software dependencies. Every time the code contained within a project is rendered by `fertile`, the package generates a file capturing the `sessionInfo()` just after the code was run. This file contains information about the R version in which the code was run, the list of packages that were loaded, and their specific versions.

The dependency information from `project_miceps` looks as follows:

```
-- Rendering R scripts... ------------------------------- fertile 0.0.0.9029 --

session_info <- "project_miceps/.software-versions.txt"
cat(readChar(session_info, 1e5))
```

The R project located at '/Users/audreybertin/Documents/thesis/index/project_miceps' was l

```
R version 4.0.2 (2020-06-22)
Platform: x86_64-apple-darwin17.0 (64-bit)
Running under: macOS Catalina 10.15.5

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRblas.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRlapack.dylib

locale:
en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
stargazer_5.2.2    skimr_2.1.2        purrr_0.3.4
ggplot2_3.3.2      tidyr_1.1.2        readr_1.4.0        dplyr_1.0.2
```

Interactively, users can access this dependency summary using the function `proj_dependency_report()`.

The `proj_pkg_script()` function builds off of this behavior, providing a method to simplify the process of installing the packages seen in the dependency report.

When run on an `R` project directory, the function creates a `.R` script file that contains the code needed to install all of the packages referenced in the project, differentiating between packages located on CRAN and those located on GitHub.

```r
install_script <- proj_pkg_script("project_miceps")
cat(readChar(install_script, 1e5))
```

```
# Run this script to install the required packages for this
R project.
# Packages hosted on CRAN...
install.packages(c( 'broom', 'dplyr', 'fs', 'ggplot2',
'purrr', 'readr', 'rmarkdown', 'skimr', 'stargazer',
'tidyr' ))
# Packages hosted on GitHub...
```

Users interested in ensuring that their project is reproducible can provide alongside it both:

1. The text-based summary of dependencies produced by `proj_dependency_report()`.
2. The `.R` dependency installation script.

Then, anyone wanting to re-create the results would have the documentation necessary to do so alongside a quick and easy method to set up the required software.

Code Commenting

In addition to README and dependency documentation, `fertile` also analyzes the quality of documentation provided directly alongside code.

Code commenting—the process of placing human-readable descriptions next to code to explain what the code is doing—is incredibly important for ensuring that a project can be understood by someone other than its author. To ensure that their code is understandable, data analysts must regularly include comments throughout their project code files.

`has_well_commented_code()` is designed to check for this. The function reads through all `.R` and `.Rmd` files in a project directory and, for each, calculates the fraction of lines in the file that are comments. Files for which comments make up less than 10% of the written lines are flagged as poorly commented, warning users to make corrections and increase the number of comments in their code.

The primary code file in `project_miceps`, "analysis.Rmd", is a great example of a poorly commented file. It only contains 6 lines of comments, while including over 100 lines of code. As such, it is flagged by `has_well_commented_code()`:

```
has_well_commented_code("project_miceps")
```

```
* Checking that code is adequately commented

  Problem: Suboptimally commented .R or .Rmd files found

  Solution: Add more comments to the files below. At least 10% of the lines should be com

  See for help: https://intelligea.wordpress.com/2013/06/30/inline-and-block-comments-in-
```

```
# A tibble: 1 x 2
  file_name                    fraction_lines_commented
  <chr>                                           <dbl>
1 project_miceps/analysis.Rmd                      0.04
```

2.1.4   Component 4: File Paths

`fertile` contains a variety of features designed to address issues with file paths. Several of these features happen proactively, warning users that they are entering a non-reproducible file path as it happens, while others can be accessed after a project has already been written.

Proactive Warnings When Coding

Proactively, `fertile` catches when an absolute path, or one leading outside of the project directory, is referenced in an input or output function and throws an error.
    This interactive behavior is one of the educational features of the package, as it immediately provides an informative message explaining to the user that such a path is not reproducible, giving the programmer an opportunity to learn and course-correct from there.
    One of the important components of R package development, however, is that a package should not interfere too much with user behavior. Users who are trying to code in their console, for example, might still want to open a file if they know the path is not reproducible if they are just trying to take a quick look at some data.
    `fertile` accounts for this. Even though the package throws an error when non-reproducible paths are used, it also provides a solution for users to override this behavior and execute the command anyway. This is done through the error messaging system. The message provided when an error is thrown is custom created, responding to the command and path which triggered the error in the first place. For example, in the code below, `read.csv("~/Desktop/my_data.csv")`, gives a different message than `read_csv("../../../Desktop/my_data.csv")`.

```
library(fertile)
file.exists("~/Desktop/my_data.csv")
```

```
[1] TRUE
```

```r
read.csv("~/Desktop/my_data.csv")
```

```
Error: Detected absolute paths. Absolute paths are not reproducible and will likely
```

```r
read_csv("../../../Desktop/my_data.csv")
```

```
Error: Detected paths that lead outside the project directory. Such paths are not re
```

In addition to catching non-reproducible file paths, `fertile` also stops the user when they try to change the working directory. Unlike with input and output functions like `read_csv()`, which can take a variety of different file paths—some which may be reproducible and some not, `setwd()` is essentially guaranteed to break reproducibility, no matter what directory it is set to. As a result, `fertile` is particularly strict with it. When a call to `setwd()` is caught, `fertile` throws an error and does not provide an option to override. Rather, the user is given an alternative function (`here::here()`) to use that achieves the same behavior in a much more reproducible way.

```r
getwd()
```

```
[1] "/Users/audreybertin/Documents/thesis/index"
```

```r
setwd("~/Desktop")
```

```
Error: setwd() is likely to break reproducibility. Use here::here() instead.
```

```r
getwd()
```

```
[1] "/Users/audreybertin/Documents/thesis/index"
```

Project Path Summaries

`fertile` also contains functionality that analyzes code for path information after it is written. For example, `proj_analyze_paths` goes through all of the paths used in a project's `.R` and `.Rmd` files and produces a report detailing which ones failed to meet reproducibility standards, explaining the problem, and providing a solution.

```r
proj_analyze_paths('project_miceps')
```

```
-- Generating reproducibility report... ----------------- fertile 0.0.0.9029 --

Checking for absolute paths...
```

```
Checking for paths outside project directory...

# A tibble: 2 x 5
  path             path_abs            func    problem       solution
  <chr>            <chr>               <chr>   <chr>          <chr>
1 /Users/audreyber~ /Users/audreybertin~ readr:~ Absolute path~ Use a relative ~
2 /Users/audreyber~ /Users/audreybertin~ readr:~ Path is not c~ Move the file a~
```

Paths can also be checked individually, or in groups, using `check_path()`.

```r
# Path inside the directory
check_path("project_miceps")
```

```
# A tibble: 0 x 3
# ... with 3 variables: path <chr>, problem <chr>, solution <chr>
```

```r
# Absolute path (current working directory)
check_path(getwd())
```

```
Error: Detected absolute paths. Absolute paths are not reproducible and will likely only w
```

```r
# Path outside the directory
check_path("../README.md")
```

```
Error: Detected paths that lead outside the project directory. Such paths are not reproduc
```

These functions, together, cover all of the reproducibility subcomponents involving file paths. Users are warned not to use absolute paths or paths pointing outside of the directory, both while they are coding and after the fact, and provided with a recommended solution to the problem if they do so. Additionally, users are prevented from using functions that will certainly break reproducibility and provided with safer alternatives.

2.1.5   Component 5: Randomness

The component of randomness is addressed by `fertile` in a reproducibility check function titled `has_no_randomness()`, which does the following: First, it checks code files to determine whether they have employed random number generation. This is confirmed by recording the random number generation seed prior to running the files, rendering the files, and then checking to see whether the seed has changed. If the seed has changed, that indicates that random number generation has occurred. Then, it determines whether whether any randomness present was semi-random (i.e. controlled and repeatable) or truly random, by checking whether the `set.seed()` function was called. If the randomness was semi-random, it is considered permissible and reproducible. Truly random code is flagged as dangerous.

   Let us consider an example from `project_miceps`. The primary code file, `analysis.Rmd` contains the following code. In this code, we see that a data set is read in via `read_csv` and reformatted. Then, the `sample_frac()` function takes a random sample of 50% of the rows of the data.

```r
mice <- read_csv("Blot_data_updated.csv") %>%
  transmute(time = `Time point`,
            sex = Sex,
            id = `Mouse #`,
            erb = `ERB Normalized Final`,
            era = `ERA Normalized Final`,
            gper = `GPER Normalized Final`) %>%
  mutate(day = case_when(
    time == 0 ~ -1,
    time == 1 ~ 0,
    time == 2 ~ 1,
    time == 3 ~ 3,
    time == 4 ~ 5,
    time == 5 ~ 7
  ),
  type = ifelse(time == 0, "control", "treatment")
  )
mice_tidy <- mice %>%
  select(-time) %>%
  gather(key = "protein", value = "amount", -day, -sex, -id, -type) %>%
  mutate(protein_label = factor(protein,
    labels = c("paste(ER, alpha)", "paste(ER, beta)", "GPER")))
```

```r
sample_frac(mice, 0.5)
```

   However, there is no seed set in the file. As a result, `has_no_randomness()` returns as a failure for the project. Had a seed been set, however, the project would have passed.

```r
has_no_randomness('project_miceps')
```

```
* Checking for no randomness

  Problem: Your code uses randomness

  Solution: Set a seed using `set.seed()` to ensure reproducibility.

  See for help: ?set.seed
```

```
# A tibble: 1 x 2
  culprit expr
  <chr>   <glue>
1 ?       Example: set.seed(1)
```

`fertile`'s randomness-centered feature helps analysts know that their use of random-
ness is controlled, ensuring that functions involving random number generation will
always produce the same output each time they are run.

2.1.6   Component 6: Readability and Style

Though not an absolutely necessary component of reproducibility, code style can have
a significant impact on how easy it is to follow the steps being taken in an analysis.
The use of consistent and highly-readable style in code greatly simplifies and speeds up
the process of understanding a data analysis project and repeating the steps included
within.

  `fertile` addresses code style via the function `has_no_lint()`. This function
checks code for compatibility with RStudio Chief Scientist Hadley Wikham's "tidy"
code format, which promotes the following best practices:

- Line length should not exceed 80 characters
- There should not be trailing whitespace
- All infix operators (`=`, `+`, `-`, `<-`, etc) should have spaces on either side
- All commas should be followed by spaces
- There is no code left in the file that is commented out
- `<-` should be used instead of `=` to assign variables
- Opening curly braces should never go on their own line and should always be
  followed by a new line
- There should always be a space between right parenthesis and an open curly
  brace
- Closing curly braces should always be on their own line, unless followed by an
  `else` statement

Any issues with incompatibility that are found by `has_no_lint()` appear in a window
by the RStudio console. This window lists the style inconsistencies found in each code
document, showing both the description of the error and the line number where it
occurred for each issue. This window is interactive; double-clicking on an error brings
the user to exact location in the code where it occurred.

  Let us consider an example. `project_miceps` contains an `RMarkdown` file titled
`analysis.Rmd`, which contains some code involved with a data analysis. Some of the
code is in tidy style but not all of it is.

  For example, line 71, part of a `ggplot` call, contains the following code:

```
# Line 71 of `project_miceps/analysis.Rmd'

geom_hline(aes(yintercept = estimate + 1.96*std.error, color = sex), linetype = 3) +
```

For this line, `has_no_lint()` finds the following inconsistencies with tidy style:

- Line 71: Lines should not be more than 80 characters.

- Line 71: Put spaces around all infix operators.

- Line 71: Trailing whitespace is superfluous.

The first comes up because the true line length, including spaces, is 84 characters. The second comes up because the `*` between `1.96` and `std.error` is not surrounded by spaces. The third is flagged because there is an empty space after the `+` at the end of the line.

An even less tidy piece of code is that found on line 189, written as follows:

```
# Line 189 of `project_miceps/analysis.Rmd'

if (length(mice) > 1){ holder_var = 1 }
```

For this line, `has_no_lint()` finds the following errors:

- Line 189: There should be a space between right parenthesis and an opening curly brace.

- Line 189: Opening curly braces should never go on their own line and should always be followed by a new line.

- Line 189: Use <-, not =, for assignment.

- Line 189: Closing curly braces should always be on their own line, unless it's followed by an else.

The first issue flagged is the fact that `){`, in the middle of the line, has no space in the middle. The second issue is flagged because the code to execute when the `if` statement is true (`holder_var = 1`) is on the same line as the `if` statement, rather than on its own line. The third error occurs because `holder_var` is defined using an `=`. And the fourth occurs because the closing curly brace is on the same line as other code rather than by itself.

These informative error messages provided by `fertile` help teach users to use consistent, tidy style in their code. They do so while making the learning process incredibly simple, allowing users to click and see exactly which symbol or character caused the error. In addition to providing a detailed explanation of the was in which the provided code is not tidy, `fertile` also suggests a simple and fast solution for resolving the inconsistencies all at once, recommending that users apply the `usethis::use_tidy_style()` function to format their code automatically.

2.1.7   Summary of Reproducibility Components

As described in the previous sections, `fertile` addresses all six major components of reproducibility via a variety of functions. Some complex components have many associated functions attached to them, while others only have one.

A summary of the six components of reproducibility and the functionalities in `fertile` associated with them is provided below.

Component	Associated Tools in *fertile*
Accessible project files	*proj_analyze_files(): returns* a report of project files (including size, type, path) *sandbox() + proj_render():* test that files are self-contained
Organized project structure	*has_proj_root() + has_no_nested_proj_root():* check for clear project structure *has_tidy_xxx():* series of functions checking for file clutter of different types *has_only_used_files():* checks that there are no excess/unused files *proj_suggest_moves() + proj_move_files():* restructure the file system
Documentation	*has_readme():* checks for the existence of a README file *has_clear_build_chain():* checks to make sure code files are clearly ordered *proj_dependency_report():* returns info on project software dependencies *proj_pkg_script():* generates an install script for dependencies *has_well_commented_code():* checks to see that code is adequately commented
File paths	*shimmed functions* (read_csv/setwd/etc) throw errors with bad paths *proj_analyze_paths():* checks that all paths in a project are reproducible *check_path():* checks a single path for reproducibility
Randomness	*has_no_randomness():* checks to make sure any randomness is controlled
Readability and style	*has_no_lint():* ensures code follows tidy style guidelines

Figure 2.3: Summary of Reproducibility Components and the Related Functionalities in 'fertile'

2.1.8   User Customizability

`fertile` does not force its users into a box, instead allowing for a great deal of user choice in terms of which aspects of reproducibility to focus on. Users can run reproducibility tests at three different scales:

1) Comprehensively, where all checks are run within a single function or two.

2) In groups, where functions focused on similar aspects of reproducibility are run together.

3) Individually, where only one reproducbility check is run at a time.

Comprehensive Features

Users who want comprehensive behavior can access it through three primary functions: `proj_check()`,`proj_analyze()`, and `proj_badges()`.

The `proj_check()` function runs all sixteen reproducibility tests in the package, noting which ones passed, which ones failed, the reason for failure, a recommended solution, and a guide to where to look for help. These tests, many of which were described in detail previously, include: looking for a clear build chain, checking to make sure the root level of the project is clear of clutter, confirming that there are no files present that are not being directly used by or created by the code, and looking for uses of randomness that do not have a call to `set.seed()` present. A full list is provided below:

```
list_checks()
```

```
-- The available checks in `fertile` are as follows: ----- fertile 0.0.0.9029 --

 [1] "has_tidy_media"         "has_tidy_images"
 [3] "has_tidy_code"          "has_tidy_raw_data"
 [5] "has_tidy_data"          "has_tidy_scripts"
 [7] "has_readme"             "has_no_lint"
 [9] "has_proj_root"          "has_no_nested_proj_root"
[11] "has_only_used_files"    "has_clear_build_chain"
[13] "has_no_absolute_paths"  "has_only_portable_paths"
[15] "has_no_randomness"      "has_well_commented_code"
```

The `proj_analyze()` function creates a report documenting the structure of a data analysis project, combining four key functions from `fertile` into one complete unit:

- `proj_analyze_packages()`, which captures a list of all packages referenced in the source code and which files they were referenced in
- `proj_analyze_files()`, which captures a list of all of the files located in the directory along with their type and size
- `proj_suggest_moves()`, which provides suggestions for how to reorganize files to create a more reproducible file structure
- `proj_analyze_paths()`, which captures a list of the non-reproducible file paths referenced in source code

```
proj_analyze('project_miceps')
```

```
-- Analysis of reproducibility for project_miceps -------- fertile 0.0.0.9029 --

--   Packages referenced in source code ------------------ fertile 0.0.0.9029 --

# A tibble: 10 x 3
   package       N used_in
   <chr>     <int> <chr>
 1 broom         1 project_miceps/analysis.Rmd
 2 dplyr         1 project_miceps/analysis.Rmd
```

```
 3 fs              1 project_miceps/analysis.Rmd
 4 ggplot2         1 project_miceps/analysis.Rmd
 5 purrr           1 project_miceps/analysis.Rmd
 6 readr           1 project_miceps/analysis.Rmd
 7 rmarkdown       1 project_miceps/analysis.Rmd
 8 skimr           1 project_miceps/analysis.Rmd
 9 stargazer       1 project_miceps/analysis.Rmd
10 tidyr           1 project_miceps/analysis.Rmd
```

```
--   Files present in directory ----------------------- fertile 0.0.0.9029 --
```

```
# A tibble: 11 x 4
   file                ext         size mime
   <fs::path>          <chr> <fs::byt> <chr>
 1 Estrogen_Receptor~  docx      10.97K application/vnd.openxmlformats-officedocu~
 2 citrate_v_time.png  png      185.41K image/png
 3 proteins_v_time.p~  png      379.94K image/png
 4 Blot_data_updated~  csv       14.43K text/csv
 5 CS_data_redone.csv  csv        7.39K text/csv
 6 mice.csv            csv       14.33K text/csv
 7 analysis.html       html       1.41M text/html
 8 README.md           md            39 text/markdown
 9 software-versions~  txt        5.34K text/plain
10 miceps.Rproj        Rproj        204 text/rstudio
11 analysis.Rmd        Rmd        5.21K text/x-markdown
```

```
--   Suggestions for moving files ---------------------- fertile 0.0.0.9029 --
```

```
# A tibble: 9 x 3
  path_rel            dir_rel     cmd
  <fs::path>          <fs::path>  <chr>
1 Blot_data_updated~  data-raw    file_move('project_miceps/Blot_data_updated.csv~
2 CS_data_redone.csv  data-raw    file_move('project_miceps/CS_data_redone.csv', ~
3 Estrogen_Receptor~  inst/other  file_move('project_miceps/Estrogen_Receptors.do~
4 analysis.Rmd        vignettes   file_move('project_miceps/analysis.Rmd', fs::di~
5 analysis.html       inst/text   file_move('project_miceps/analysis.html', fs::d~
6 citrate_v_time.png  inst/image  file_move('project_miceps/citrate_v_time.png', ~
7 mice.csv            data-raw    file_move('project_miceps/mice.csv', fs::dir_cr~
8 proteins_v_time.p~  inst/image  file_move('project_miceps/proteins_v_time.png',~
9 software-versions~  inst/text   file_move('project_miceps/software-versions.txt~
```

```
--   Problematic paths logged ------------------------- fertile 0.0.0.9029 --
```

```
NULL
```

Similar to `proj_check()`, `proj_badges()` runs all sixteen reproducibility checks—`has_tidy_media`, `has_readme`, `has_no_absolute_paths`, etc. However, the way the function displays the resulting information is different. Rather than report the results of each check individually alongside detailed information about why each failed, `proj_badges()` provides a higher level summary, reporting on the *groups* of checks that either passed or failed.

In `proj_badges()`, the reproducibility checks are grouped into six categories corresponding to the six major components of reproducibility. These categories, alongside the checks which belong to them, are listed below:

- Project Structure

    - `has_proj_root()`
    - `has_no_nested_proj_root()`

- Tidy Files

    - `has_tidy_media()`
    - `has_tidy_images()`
    - `has_tidy_code()`
    - `has_tidy_raw_data()`
    - `has_tidy_data()`
    - `has_tidy_scripts()`
    - `has_only_used_files()`

- Documentation

    - `has_readme()`
    - `has_clear_build_chain()`
    - `has_well_commented_code()`

- File Paths

    - `has_no_absolute_paths()`
    - `has_only_portable_paths()`

- Randomness

    - `has_no_randomness()`

- Code Style

    - `has_no_lint()`

When users pass all of the checks associated with one of the categories, they receive a badge certifying their success in that area. If any fewer than *all* available checks in a category are passed, then the user fails that category.
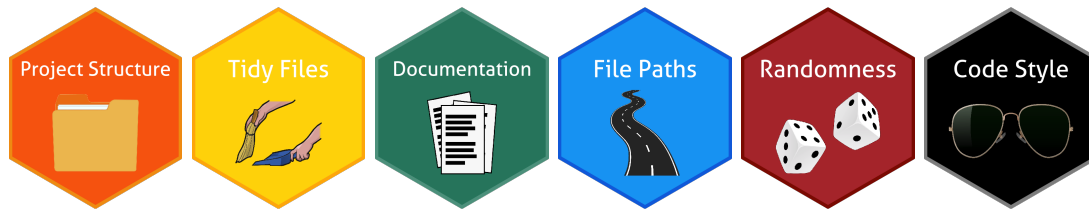
Figure 2.4: The 6 Reproducibilty Badges Available in 'fertile'

`proj_badges()` identifies the badges that a data analysis project should receive and then outputs a summary of this information—in the form of an `html` document—to the RStudio `Viewer` pane. This document—designed to produce a summary of a project that can be easily shared in places outside of `RStudio` (unlike the summaries produced by most other functions, which just produce output in the `R` console)—contains the following primary pieces of information:

1. The name of the folder where the project is stored.
2. The reproducibility badges awarded to the project.
3. The reproducibility badges failed by the project.
4. If failures occurred, the reasons why. These are broken down by badge and list the checks associated with that badge that did not pass.
5. A date/time/timezone stamp of when the `html` summary was created.
6. Information about the user who generated the `html` summary (their full name, computer username, email, and GitHub username, if available).
7. Information about the `R` version, platform, and operating system of the computer used to generate the `html` at the time that it was created.
8. The names of files in the project that the `html` found the badges for as well as their date of last modification.

The `html` output for `project_miceps` can be seen below:

Figure 2.5: Output: Badges and Reasons for Failure

Code Style

```
## # A tibble: 1 x 1
##   check_name
##   <chr>
## 1 has_no_lint
```

Output Generation Details:

This project summary was generated on 2020-10-29 at 14:00:51 (America/New_York) by a user with the following information:

- Full name: Audrey Bertin
- Username: audreybertin
- Email: N/A
- GitHub Username: N/A

The computer used to generate this file was running R version 4.0.2 (2020-06-22) on the x86_64-apple-darwin17.0 (64-bit) platform and the macOS Catalina 10.15.5 operating system.

The files analyzed in the creation of this summary, as well as their last-modified timestamp, are provided below:

```
## # A tibble: 9 x 2
##   file_name            last_edited
##   <chr>                <dttm>
## 1 Blot_data_updated.csv    2020-10-12 14:25:17
## 2 CS_data_redone.csv       2020-10-12 14:25:22
## 3 Estrogen_Receptors.docx  2020-10-12 14:25:27
## 4 README.md                2019-01-25 14:19:39
## 5 analysis.Rmd             2020-10-12 14:25:32
## 6 citrate_v_time.png       2020-10-29 14:00:40
## 7 mice.csv                 2020-10-12 14:29:31
## 8 miceps.Rproj             2019-01-25 14:19:39
## 9 proteins_v_time.png      2020-10-29 14:00:39
```

Figure 2.6: Output: System/User Information and File List

Together, `proj_check()`,`proj_analyze()`, and `proj_badges()` cover a significant portion of all six major components of reproducibility.

Features Allowing For Summary By Group

Users wanting to focus on groups of checks can do so using the `proj_check_some()` function. `proj_check_some()` leverages helper functions from `tidyselect` (Henry & Wickham (2020)) to allow users to call groups of similar checks together. `tidyselect` helpers tell `fertile` to call only the checks whose names meet certain conditions. The helper functions that can be passed to `proj_check_some()` are:

- `starts_with()`: Starts with a prefix.
- `ends_with()`: Ends with a suffix.
- `contains()`: Contains a literal string.
- `matches()`: Matches a regular expression.

For example, a variety of checks in `fertile` focus on making sure the project has a "tidy" structure–essentially that there are not files cluttered together all in one folder. Users interested in checking their tidyness can do so all at once using `proj_check_some()` as follows:

```
proj_check_some("project_miceps", contains("tidy"))
```

```
-- Running reproducibility checks ---------------------- fertile 0.0.0.9029 --
```

v Checking for no *.R scripts at root level

v Checking for no *.rda files at root level

v Checking for no A/V files at root level

* Checking for no image files at root level

  Problem: Image files in root directory clutter project

  Solution: Move source files to img/ directory

  See for help: ?fs::file_move

```
# A tibble: 2 x 2
  culprit                    expr
  <fs::path>                 <glue>
1 project_miceps/citrate_v_t~ fs::file_move('project_miceps/citrate_v_time.png'~
2 project_miceps/proteins_v_~ fs::file_move('project_miceps/proteins_v_time.png~
```

* Checking for no raw data files at root level

  Problem: Raw data files in root directory clutter project

  Solution: Move raw data files to data-raw/ directory

  See for help: ?fs::file_move

```
# A tibble: 3 x 2
  culprit                    expr
  <fs::path>                 <glue>
1 project_miceps/Blot_data_u~ fs::file_move('project_miceps/Blot_data_updated.c~
2 project_miceps/CS_data_red~ fs::file_move('project_miceps/CS_data_redone.csv'~
3 project_miceps/mice.csv     fs::file_move('project_miceps/mice.csv', here::he~
```

v Checking for no source files at root level

-- Summary of fertile checks --------------------------- fertile 0.0.0.9029 --

v Reproducibility checks passed: 4

* Reproducibility checks to work on: 2

* Checking for no image files at root level

  Problem: Image files in root directory clutter project

  Solution: Move source files to img/ directory

```
   See for help: ?fs::file_move

# A tibble: 2 x 2
  culprit                  expr
  <fs::path>               <glue>
1 project_miceps/citrate_v_t~ fs::file_move('project_miceps/citrate_v_time.png'~
2 project_miceps/proteins_v_~ fs::file_move('project_miceps/proteins_v_time.png~

* Checking for no raw data files at root level

   Problem: Raw data files in root directory clutter project

   Solution: Move raw data files to data-raw/ directory

   See for help: ?fs::file_move

# A tibble: 3 x 2
  culprit                  expr
  <fs::path>               <glue>
1 project_miceps/Blot_data_u~ fs::file_move('project_miceps/Blot_data_updated.c~
2 project_miceps/CS_data_red~ fs::file_move('project_miceps/CS_data_redone.csv'~
3 project_miceps/mice.csv     fs::file_move('project_miceps/mice.csv', here::he~
```

Users might also attempt to call the two checks involving project roots together:

```
proj_check_some("project_miceps", ends_with("root"))
```

```
-- Running reproducibility checks ---------------------- fertile 0.0.0.9029 --

v Checking for nested .Rproj files within project

v Checking for single .Rproj file at root level

-- Summary of fertile checks --------------------------- fertile 0.0.0.9029 --

v Reproducibility checks passed: 2
```

Or, perhaps, they might want to run all the functions beginning with "has_only":

```
proj_check_some("project_miceps", starts_with("has_only"))
```

```
-- Compiling... ---------------------------------------- fertile 0.0.0.9029 --

-- Rendering R scripts... ------------------------------ fertile 0.0.0.9029 --

-- Running reproducibility checks ---------------------- fertile 0.0.0.9029 --
```

```
Joining, by = "path_abs"

* Checking for only portable paths

   Problem: Non-portable paths won't necessarily work for others

   Solution: Use relative paths.

   See for help: ?fs::path_rel

# A tibble: 1 x 2
  culprit                                expr
  <fs::path>                             <glue>
1 /Users/audreybertin/Documents/thesis/in~ fs::path_rel('/Users/audreybertin/Do~

* Checking to see if all files in directory are used in code

   Problem: You have files in your project directory which are not being used.

   Solution: Use or delete files.

   See for help: ?fs::file_delete

# A tibble: 2 x 1
  path_abs
  <chr>
1 /Users/audreybertin/Documents/thesis/index/project_miceps/Estrogen_Receptors.~
2 /Users/audreybertin/Documents/thesis/index/project_miceps/mice.csv

-- Summary of fertile checks ---------------------------- fertile 0.0.0.9029 --

v Reproducibility checks passed: 0

* Reproducibility checks to work on: 2

* Checking for only portable paths

   Problem: Non-portable paths won't necessarily work for others

   Solution: Use relative paths.

   See for help: ?fs::path_rel

# A tibble: 1 x 2
  culprit                                expr
  <fs::path>                             <glue>
1 /Users/audreybertin/Documents/thesis/in~ fs::path_rel('/Users/audreybertin/Do~
```

```
* Checking to see if all files in directory are used in code

  Problem: You have files in your project directory which are not being used.

  Solution: Use or delete files.

  See for help: ?fs::file_delete
```

```
# A tibble: 2 x 1
  path_abs
  <chr>
1 /Users/audreybertin/Documents/thesis/index/project_miceps/Estrogen_Receptors.~
2 /Users/audreybertin/Documents/thesis/index/project_miceps/mice.csv
```

Individual Checks

If users do not want to run functions in groups, and prefer to run them individually, that option is also provided to them. Every check that makes up `check()` and every subcomponent of `project_analyze()`can be run individually.

2.1.9   Educational Features

Simply noting and correcting issues with reproducibility is not enough to produce lasting change in the scientific community. Data analysts and software users must also be educated on why their choices were not reproducible so that they do not fall victim to those mistakes again in the future, but also so that they can share their knowledge and experience with others in the scientific community.

  `fertile` prioritizes this idea of reproducibility education throughout many of its functionalities.

  One of the major ways through which `fertile` educates is users is a system of command tracking and interactive messaging. As long as `fertile` is loaded in `R`, the package records when commands that have the potential to affect reproducibility are run in the console.

  As soon as a dangerous function is called, `fertile` alerts the user of their mistake and provides suggestions for alternative solutions. This behavior, explained in more detail in Chapter 2.2.4, gives users immediate feedback on their behavior. In addition to assisting users in the moment, this method has also been shown to increase long-run retention of information when compared with feedback after the fact (M. L. Epstein et al. (2002)).

  Users interested in looking back at their past choices and mistakes can do so as well. The `log_report()` function provides access to a log listing the commands with a link to reproducibility that have been called.

`log_report``()`

```
Reading from /Users/audreybertin/Documents/thesis/index/.fertile_log.csv
```

```
# A tibble: 593 x 4
   path              path_abs                            func      timestamp
   <chr>             <chr>                               <chr>     <dttm>
 1 package:remotes   <NA>                                base::re~ 2020-09-20 15:20:34
 2 package:thesis~   <NA>                                base::re~ 2020-09-20 15:20:34
 3 package:thesis~   <NA>                                base::li~ 2020-09-20 15:20:34
 4 package:stringr   <NA>                                base::li~ 2020-09-21 13:28:29
 5 package:fertile   <NA>                                base::li~ 2020-09-21 13:28:29
 6 ~/Desktop/my_d~   ~/Desktop/my_data.csv               utils::r~ 2020-09-21 13:28:29
 7 ../../../Deskt~   /Users/audreybertin/Desktop/my~     utils::r~ 2020-09-21 13:28:29
 8 package:purrr     <NA>                                base::li~ 2020-09-21 13:28:36
 9 package:forcats   <NA>                                base::li~ 2020-09-21 13:28:36
10 project_miceps~   /Users/audreybertin/Documents/~     readr::r~ 2020-09-21 13:28:36
# ... with 583 more rows
```

Depending on how much history they want to keep track of, users have the option to clear the log and start from scratch via the function `log_clear()`.

In addition to this educational logging behavior, the reproducibility checks contained in the `check()` function also include educational messages. We consider several examples below:

- `has_no_randomness()`, when it fails, tells users that their code uses randomness and they should use the `set.seed()` function to control it.

- `has_only_used_files()`, when it fails, tells users that there are files present in the directory not being used for any purpose and provides a function (`fs::file_delete`) to use in order to remove them.

- `has_no_lint()`, when it fails, provides users a list of all of the ways in which their code fails to follow tidy style and points them to the exact lines and characters in the code where the mistakes occurred.

- `has_tidy_images`, when it fails, tells users that it has found files in the root directory which add clutter and recommends that they be moved to an `img/` directory.

Users are provided with an informative message about their issue but are not always provided a fully automated solution. This behavior encourages them to learn as they have to execute the suggested solution themselves.

However, even though `fertile` often requires users to take their own actions, that does not mean that the package requires users to be of high skill level to use. `fertile` is designed in such a way so its educational benefits can be achieved with relative simplicity and minimal effort so that even users brand new to `R` can gain knowledge and awareness of reproducibility from using it.

For example, the interactive messaging features require no effort beyond the loading of the package with `library(fertile)` to activate. And many of the educational benefits contained within the checks can be gained all at once as well, with a single

call of the `check()` function. Users looking for more customizability have the option to go into more detail with the more complex functions like `sandbox()`, but they are not necessary for users to gain benefit from the package.

2.2   How `fertile` Works

In the world of `R` packages, `fertile`'s behavior is rather unusual. Almost no other packages manipulate their users' `R` environments or file structure, instead remaining relatively self-contained. This unique behavior necessitates the use of several non-traditional techniques, described below.

2.2.1   Shims

Much of the functionality in `fertile` is achieved by writing `shims`. In their application to `fertile`, shims can be defined as functions that transparently intercept users' intended actions and slightly alter their execution.

`fertile` contains shims for a variety of common functions that may affect reproducibilty, including those that read and write files, load libraries, and set random number generation seeds. A full list of shimmed functions, organized by their original package, is provided below:

Package	Function(s)
dplyr	tbl
readr	read_csv, read_csv2, read_delim, read_file, read_file_raw, read_fwf, read_lines, read_lines_raw, read_log, read_table, read_table2, read_tsv, write_csv
ggplot2	ggsave
stats	read.ftable
utils	read.csv, read.csv2, read.delim, read.delim2, read.DIF, read.fortran, read.fwf, read.table, write.csv
base	library, load, read.dcf, require, save, set.seed, setwd, source
readxl	read_excel
rjson	fromJSON
foreign	read.dta, read.mtp, read.spss, read.systat
sas7bdat	read.sas7bdat

Figure 2.7: List of Functions Shimmed by 'fertile'

When users perform actions that may threaten reproducibility, the package's shimmed functions intercept the user's commands and perform various logging and

checking tasks before executing the desired function.

This allows `fertile` to warn users when they make mistakes and also to keep track of past behavior via a log of previously entered commands.

The process for writing a shim is as follows:

1. Identify an `R` function that is likely to be involved in operations that may break reproducibility. Popular functions associated with only one package (e.g., `read_csv()` from `readr`) are ideal candidates.

2. Create a function in `fertile` with the same name that takes the same arguments (and always the dots ...).

3. Write this new function so that it:

a) captures any arguments,
b) logs the name of the function called,
c) performs any checks on these arguments, and
d) calls the original function with the original arguments. Except where warranted, the execution looks the same to the user as if they were calling the original function.

Most shims, when written, are relatively simple. Several, such as that for `library()` are more complex, but many follow the same basic format that can be seen in this example for `read_csv()`:

```
fertile::read_csv
```

```
function(file, ...) {

  if (interactive_log_on()) {
    if (Sys.getenv("FERTILE_RENDER_MODE") == TRUE){
      log_push('Seed Before', .Random.seed[2])
    }

    log_push(file, "readr::read_csv")

    check_path_safe(file, ... = "readr::read_csv")

    data <- readr::read_csv(file, ...)

    if (Sys.getenv("FERTILE_RENDER_MODE") == TRUE){
      log_push('Seed After', .Random.seed[2])
    }

    return (data)
}
```

```
}
<bytecode: 0x7f9d2c7af7d0>
<environment: namespace:fertile>
```

This functionality all occurs without the knowledge of the user. Consider the example of `read_csv()`. `read_csv()` is a very popular function from the `readr` package for reading in data files. Users with both `readr` and `fertile` loaded, will experience the following. The user will call `read_csv()` as normal, thinking that they are calling `readr::read_csv()`. However, they will actually be calling `fertile::read_csv()`, a very similar function with the same name. `fertile::read_csv()` will then capture the file path the user provided and check whether it is reproducible. If it is deemed okay, the function will execute as intended and read in the data just as `readr::read_csv()` would. If it is deemed non-reproducible, the function will return an error telling the user to use an alternate file path. Either way, `fertile` will record that the user called `read_csv()` and note the path that was provided to it for future reference.

This behavior, however, is dependent `fertile` remaining at the top of the `search()` path so that its functions are called preferentially over the original functions that it has shimmed. In order to ensure that the `fertile` versions of functions ("shims") always supersede ("mask") their original namesakes when called, `fertile` uses its own shims of the `library` and `require` functions to manipulate the `R search` path so that it is always located in the first position.

In the `fertile` version of `library()`, `fertile` is detached from the search path, the requested package is loaded, and then `fertile` is reattached. This ensures that when a user executes a command, `R` will check `fertile` for a matching function before considering other packages.

2.2.2   Hidden Files

In order to store and analyze information about user behavior and code structure, `fertile` utilizes three different types of hidden files, two of which are `.csv` format and one of which is a text file. The hidden `.csv` files for `project_miceps` can be seen below:

```
fs::dir_ls("project_miceps", all = TRUE, glob = "*_log.csv")
```

```
project_miceps/.fertile_log.csv        project_miceps/.fertile_render_log.csv
```

The interactive log (`.fertile_log.csv`), accessible via `log_report()`, is created as soon as a user executes their first piece of code that could threaten reproducibility. This file tracks all of the shimmed functions executed by the user, either in the console or when running code chunks by hand (rather than knitting a file). It reports the function called, the relevant argument passed in (either a file path or `R` package name), and the time stamp of when the function was executed. Users can clear the data from this file and start fresh at any time with `log_clear()`.

The render log (`.fertile_render_log.csv`), accessible via `render_log_report()`, has a similar structure to the interactive log but is not under the control of the user.

It tracks information about the code contained within `.R` and `.Rmd` files within the project a user is testing for reproducibility. A new render log file can be generated in one of three different ways:

1. The first time a user runs one of the major reproducibility checks from `fertile`, such as `proj_check()`, `proj_analyze()`, or one of the smaller checks within that requires access to the contents of code files.

2. Any time a check involving code is called in `fertile`, the package checks to see whether any code files have been updated since the last time a render log was generated. If so, a new render log is generated.

3. The user can generate a new file manually at any time by executing the `proj_render()` command.

The render log contains information used to run many of the checks in `fertile`. It captures the random number generator seed before and after executing code, notes which packages and files are accessed and the function with which they were called, and contains a timestamp of the last time the code was run by `fertile`. Users cannot erase it easily.

```
render_log_report("project_miceps")
```

Reading from /Users/audreybertin/Documents/thesis/index/project_miceps/.fertile_rend

```
# A tibble: 21 x 4
   path                 path_abs                func     timestamp
   <chr>                <chr>                   <chr>    <dttm>
 1 Seed @ Start         <NA>                    599      2020-11-02 23:55:08
 2 package:dplyr        <NA>                    base::~  2020-11-02 23:55:09
 3 package:readr        <NA>                    base::~  2020-11-02 23:55:09
 4 package:tidyr        <NA>                    base::~  2020-11-02 23:55:09
 5 package:ggplot2      <NA>                    base::~  2020-11-02 23:55:09
 6 package:purrr        <NA>                    base::~  2020-11-02 23:55:09
 7 Seed Before          <NA>                    599      2020-11-02 23:55:09
 8 /Users/audreybertin/Do~ /Users/audreybertin/Docu~ readr::~ 2020-11-02 23:55:09
 9 Seed After           <NA>                    613      2020-11-02 23:55:09
10 Seed Before          <NA>                    613      2020-11-02 23:55:09
# ... with 11 more rows
```

These files, integral to many of the functionalities in `fertile`, are not visible when looking at the file system. Users can only access them with functions provided by fertile, and user permissions are quite restrictive. Except for removing the command history in the interactive log, users cannot modify their contents, and neither the render log nor the interactive log can be deleted without the user modifying their file system. This prevents the files from mistakenly being tampered with, potentially impairing

their functionality, and ensures that `fertile` always retains accurate information of user behavior.

In addition to the two log files, `fertile` keeps a third hidden file to track project dependencies. Described in detail in the section on documentation, this file keeps track of the software setup that a project is run under. A new version is generated every time `fertile` compiles the project code files or when users request to view the file (via a special access function). Like the log files, however, there is no simple way to manually modify or delete the file, ensuring that it does not accidentally get changed in a way that would damage reproducibility.

2.2.3   Environment Variables

The shims in `fertile` are designed to be able to write to both the interactive and render logs. That way, no matter whether a user calls `read_csv()` interactively or writes it in their code file, `fertile` will still take note of the fact that the action has happened.

Given the different purposes of each file, however, it is important that `fertile` be able to identify when a function execution should be saved to the interactive log versus when it should be saved to the render log.

This information is tracked via a logical environment variable: `FERTILE_RENDER_MODE`. When `FERTILE_RENDER_MODE` is `TRUE`, executed shims are saved to the render log. When it is `FALSE`, they are saved to the interactive log.

Since `fertile` is designed to always be capturing information about interactive user behavior, `FERTILE_RENDER_MODE` is `FALSE` by default.

It is only changed to `TRUE` when `fertile` is executing functions that involve the rendering of `.R` and `.Rmd` code files. At the start of all such functions, `fertile` sets the environment variable to `TRUE`, executes the majority of the function, and then sets it back to `FALSE` before exiting. This ensures that as soon as the function has finished running, all new commands get executed on the interactive log, rather on the render log which was just generated.

The example of `has_only_portable_paths()` illustrates this functionality. We see several clear steps to this function:

1. The environment variable is set to `TRUE`, so that `fertile` knows to write any information about shimmed functions to the render log.

2. If a render log does not yet exist or the project has been updated since the last time one was generated, then the project is rendered with `proj_render()`, generating a new render log.

3. This render log is read to find information about the file paths that were captured when executing the code files.

4. `fertile` checks to see if the paths are portable and outputs a a list of the ones that are not, if any, in addition to some information about how to correct that.

5. The environment variable is set back to **FALSE** so that interactive behavior is
once again captured.

has_only_portable_paths

```r
function(path = ".") {

  Sys.setenv("FERTILE_RENDER_MODE" = TRUE)

  check_is_dir(path)

  if (!has_rendered(path)) {
    proj_render(path)
  }

  paths <- log_report(path) %>%
    dplyr::filter(!grepl("package:", path)) %>%
    dplyr::pull(path)

  good <- paths %>%
    is_path_portable()

  errors <- tibble(
    culprit = as_fs_path(paths[!good]),
    expr = glue("fs::path_rel('{culprit}')")
  )

  Sys.setenv("FERTILE_RENDER_MODE" = FALSE)

  make_check(
    name = "Checking for only portable paths",
    state = all(good),
    problem = "Non-portable paths won't necessarily work for others",
    solution = "Use relative paths.",
    help = "?fs::path_rel",
    errors = errors
  )

}
<bytecode: 0x7f9d334aecc8>
<environment: namespace:fertile>
attr(,"req_compilation")
[1] TRUE
```

2.2.4   The Dots (. . .)

`fertile` utilizes the dots (. . .), which allow a function to accept additional arguments beyond those pre-defined in the function, to facilitate much of its behavior. The two primary locations the dots are used are in shims, the file path messaging system, and the `proj_check_some()` function.

Many of the shimmed functions in `fertile` accept a large number of arguments. Consider `read_csv`. The function requires only one argument (`file`) but allows for up to 14.

```
readr::read_csv
```

```
function (file, col_names = TRUE, col_types = NULL, locale = default_locale(),
    na = c("", "NA"), quoted_na = TRUE, quote = "\"", comment = "",
    trim_ws = TRUE, skip = 0, n_max = Inf, guess_max = min(1000,
        n_max), progress = show_progress(), skip_empty_rows = TRUE)
{
    tokenizer <- tokenizer_csv(na = na, quoted_na = quoted_na,
        quote = quote, comment = comment, trim_ws = trim_ws,
        skip_empty_rows = skip_empty_rows)
    read_delimited(file, tokenizer, col_names = col_names, col_types = col_types,
        locale = locale, skip = skip, skip_empty_rows = skip_empty_rows,
        comment = comment, n_max = n_max, guess_max = guess_max,
        progress = progress)
}
<bytecode: 0x7f9d2aa0c160>
<environment: namespace:readr>
```

When users call the shimmed version of `read_csv`, `fertile` does not need to process any arguments other than `file`, since that is the only piece of information directly relevant to reproducibility. Instead of defining all of the arguments once again, `fertile`'s `read_csv` is written to accept a file name and the dots , which then capture and save the additional input provided by the user. `fertile` checks the file path, and then uses the saved dots to then execute `readr::read_csv` the way that the user originally intended.

```
fertile::read_csv
```

```
function(file, ...) {

  if (interactive_log_on()) {
    if (Sys.getenv("FERTILE_RENDER_MODE") == TRUE){
      log_push('Seed Before', .Random.seed[2])
    }
```

```
    log_push(file, "readr::read_csv")

    check_path_safe(file, ... = "readr::read_csv")

    data <- readr::read_csv(file, ...)

    if (Sys.getenv("FERTILE_RENDER_MODE") == TRUE){
      log_push('Seed After', .Random.seed[2])
    }

    return (data)
  }
}
<bytecode: 0x7f9d2c7af7d0>
<environment: namespace:fertile>
```

The majority of shims have a similar structure. Since almost all shimmed functions in `fertile` take a large number of arguments, the `fertile` versions utilize the dots to simplify the process of capturing this user input and saving it for execution later.

The dots are also used in the interactive file path messaging system, which notifies the user when they reference an absolute path or one leading outside of their project directory by throwing an error and providing an informative warning message. They are used in combination with shims to capture information about the user action that caused the error and to pass this information to the messaging system. This ensures that the override code provided at the end of the error message is customized to the user's behavior. In the example below, the warning message recognizes that `read.csv` was called with the path `~/Desktop/my_data.csv`, and incorporates that into the suggested code:

```
read.csv("~/Desktop/my_data.csv")
```

```
    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
1  21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
2  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
3  22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
4  21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
5  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
6  18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
7  14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
8  24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
9  22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
10 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
11 17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
12 16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
13 17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
```

```
14 15.2    8 275.8 180 3.07 3.780 18.00  0  0    3    3
15 10.4    8 472.0 205 2.93 5.250 17.98  0  0    3    4
16 10.4    8 460.0 215 3.00 5.424 17.82  0  0    3    4
17 14.7    8 440.0 230 3.23 5.345 17.42  0  0    3    4
18 32.4    4  78.7  66 4.08 2.200 19.47  1  1    4    1
19 30.4    4  75.7  52 4.93 1.615 18.52  1  1    4    2
20 33.9    4  71.1  65 4.22 1.835 19.90  1  1    4    1
21 21.5    4 120.1  97 3.70 2.465 20.01  1  0    3    1
22 15.5    8 318.0 150 2.76 3.520 16.87  0  0    3    2
23 15.2    8 304.0 150 3.15 3.435 17.30  0  0    3    2
24 13.3    8 350.0 245 3.73 3.840 15.41  0  0    3    4
25 19.2    8 400.0 175 3.08 3.845 17.05  0  0    3    2
26 27.3    4  79.0  66 4.08 1.935 18.90  1  1    4    1
27 26.0    4 120.3  91 4.43 2.140 16.70  0  1    5    2
28 30.4    4  95.1 113 3.77 1.513 16.90  1  1    5    2
29 15.8    8 351.0 264 4.22 3.170 14.50  0  1    5    4
30 19.7    6 145.0 175 3.62 2.770 15.50  0  1    5    6
31 15.0    8 301.0 335 3.54 3.570 14.60  0  1    5    8
32 21.4    4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Finally, the dots play a big role in the `proj_check_some` function. Recall that `proj_check_some` allows users to run a selection of `fertile`'s checks by calling a `tidyselect` helper to pull out a subset of checks with names matching a certain definition.

This function, which accepts the arguments "(path,...)," operates by allowing the users to pass in their `tidyselect` call to the dots. The list of available checks are converted into the columns of a dataframe, then passed through `dplyr::select(...)`, where the dots contain the information about the user's `tidyselect` call. Then, all of the checks matching the `tidyselect` call are run on the provided directory path.

2.2.5   Parameterized Reports

The `proj_badges()` function builds an `html` document based on a collection of information calculated when rendering an `R` project, then displays that `html` document in the `Viewer` window of `RStudio`.

This `html` document is customized based on both the computer and the directory that `proj_badges()` was run on:

Mac and Windows users will receive different outputs, as will those who have the same type of computer but are using a a different `R` version or operating system (e.g. macOS Catalina 10.15.7 vs macOS Mojave 10.14.6).

The output of `proj_badges()` is also dependent on the files included in the directory it analyzes. When run on files with different content, names, and/or metadata, the function can produce reports showing different sets of passed/failed badges, different explanations for why a project failed certain badges, and a different list summarizing the names/modification dates of the files used to generate the project

summary.

This customization is achieved through the use of an `RMarkdown Parameterized Report`. Parameterized reports allow for the creation of customized outputs based on different sets of inputs (parameters), all using the same basic template document.

Within `fertile`, there is an `RMarkdown` template that contains a variety of defined parameters. These include, among other things:

1. Vectors containing the names of badges that were awarded and of those that were failed, `2.tibbles` containing information why the failed badges were not achieved (the names of the badge-specific tests that failed), and
2. A `tibble` containing information about the names and last-modified dates of files in the project directory of interest.

Although it always follows the same basic structure defined in the template, the appearance of the knit output document varies based on the value of these parameters.

This functionality, along with the other methods of shims, hidden files, environment variables, and the dots helps improve the user experience of `fertile`. These unconventional techniques allow for the reliable tracking of user behavior behind the scenes and provide functions, like `proj_check_some`, that are intuitive for the user to operate.

2.3 Summary

Recall the list of features necessary for an effective reproducibility tool that were defined at the end of Chapter 1. Many of the tools and teaching methods considered previously fail to meet these standards: they are often complicated and/or narrow in scope. `fertile`, on the other hand, has none of these problems. Rather, the package meets all of the defined conditions for effectiveness:

1) Be simple, with a small library of functions/tools that are straightforward to use.

`fertile` is very simple to use. Most of the package's features can be achieved with just two functions: `proj_analyze()` and `proj_check()`. The the interactive warning features—which throw errors when users reference absolute paths or those leading outside of the project directory—are even more straightforward. As soon as `fertile` is loaded, they are automatically enabled and require no additional effort from the user. Additionally, the operation of the functions themselves is quite simple—most functions in the package require only one argument: the path to the directory that the user wants to run the function on.

2) Be accessible to a variety of users, with a relatively small learning curve.

`fertile` has very few barriers to entry. The package only requires that its users be familiar with installing and loading packages from GitHub, executing functions in the console, and creating R projects, all of which can be learned in a handful of quick web searches. Running `proj_check_some()` successfully would also require knowledge of the `tidyselect` helpers (`starts_with()`, `contains()`, etc.), but that function is

not necessary for use in any way. Users who were unfamiliar with its behavior could simply choose just to run a handful of individual functions instead and achieve the same results.

** 3) Be able to address a wide variety of aspects of reproducibility, rather than just one or two key issues.**

`fertile` contains functions that address all six primary components of reproducibility (see Fig. 2.2). Some components, which are more complicated and have a variety of smaller parts—such as documentation—have more functions associated with them. Randomness, the simplest component, has just one. While there are a variety of different functions for each component, it is not necessary for them to all be run independently. If a user is interested in checking all six components simultaneously, running `proj_analyze()`, `proj_check()`, and `proj_dependency_report()` together will achieve this goal.

4) Have features specific to a particular coding language that can address that language's unique challenges.

`fertile` contains features to address the package system in R.

R is an open-source software that relies on packages (collections of functions) to achieve much of its functionality. Users who are looking to expand the available functions can contribute software packages to the community for public use. These packages are typically hosted either on the Comprehensive R Archive Network (CRAN) or on GitHub. Once a package is available on either site, any R user can download it and install it on their local R version.

Most data analyses in R are dependent on a variety of packages. Depending on the complexity and topic of the analysis, the exact number may vary, but most analyses rely on at least a handful to operate successfully.

The challenge with this system is that R packages are updated constantly by their creators. Users who go a few weeks without updating their software might find that dozens of their packages have updates available. Due to the frequency of updates, it is a common occurrence for code that once worked to stop functioning due to a change in the functionality of one of the packages it is dependent on.

As a result of this, tracking dependencies in R is more important than in some other coding languages. `fertile` attempts to address this through its dependency-tracking features (`proj_pkg_script()` and `proj_dependency_report()`), which together help the user of a package record the exact package versions that their project is dependent on and simplify the process of installing these dependencies by identifying which came from GitHub and which came from CRAN and creating a script with which to install them.

Another R-specific feature that `fertile` addresses is randomness. Randomness is incredibly important in statistics. Many statistical methods rely on random sampling in some way, and as a result, R—due to its purpose as a statistics-specific coding language—contains a wide variety of functions that use random number generation. In their default states, these functions produce a different result each time, and as a result are an inherent threat to reproducibility. In order to account for this, `fertile` contains the `has_no_randomness()` function, which reads scripts to ensure that any randomness (if present), is controlled (made pseudo-random) and reproducible.

5) Be customizable, allowing users to choose for themselves which aspects of reproducibility they want to focus on.

As discussed in Chapter 2.1.8, `fertile` is highly customizable, providing users with a wide variety of options for how they can run the package's reproducibility tests. The main reproducibility checks can be run all at once (using `proj_check()`), in groups (using `proj_check_some()`), or individually. Users who only want basic information can rely on only `proj_analyze()` and `proj_check()`, while those looking for more advanced reproducibility information can delve into the project dependency functions, as well as `sandbox()` + `proj_render()`, to ensure that projects are completely self contained.

6) Be educational, teaching those that use it about why their projects are not reproducible and how to correct that in the future.

`fertile` contains many different educational features. All of the reproducibility checks, when they fail, produce an informative warning message detailing where the failure occurred and providing a solution for how to address it. Additionally, the interactive warning system plays a big role in reproducibility education. Users are stopped immediately any time they use a non-reproducible file path and told why their path is problematic. This helps educate users on the correct use of file paths, ensuring that such mistakes do not happen many times.

7) Be applicable to a wide variety of domains.

Due to its low learning curve, customizability, and all-component-encompassing nature, `fertile` has the potential to provide benefits to users in a wide variety of donains. We will consider these in detail in the next chapter.

Chapter 3

Incorporating `fertile` Into The Greater Data Science Community

Finding a solution to addressing reproducibility on a widespread scale is a challenging problem. Attempts to do so–in academic publishing, software, and data science education–have made some progress, but many solutions have significant flaws. Primarily, they either:

A) Only address one small aspect of reproducibility–for example, software that focuses on version control or a set of journal guidelines requesting only that code and data be provided, but giving no further detail.

<div align="center">OR</div>

B) Are challenging, time consuming, and/or burdensome to implement–for example, extensive journal guidelines, complex software packages with confusing functions, or academic courses on reproducibility that are only accessible to masters' students and take time away from other topics.

`fertile` is an attempt to address reproducibility in a way that does not fall victim to either of these challenges. Rather than focus on one area of expertise, `fertile` contains features focused on each of the six major components of reproducibility. Its self-contained nature allows users to address all aspects of reproducibility in one package; users can achieve near- or complete reproducibility with just a single piece of software.

`fertile` also makes the processes of both achieving *and* checking reproducibility simple and fast. Those looking to check whether a project is reproducible can almost instantaneously receive a full report of where the project succeeds and where it fails, and those looking to improve their reproducibility can receive and act on `fertile`'s clear suggestions with minimal effort. Some of the package's features are enabled automatically and most others can be accessed with only a handful of functions, all of which are very simple in function.

Additionally, `fertile` does not just provide a report on reproducibility and leave it at that. Instead, it attempts to teach its users the concepts of reproducibility in the

same way that reproducibility-focused classes are meant to do. Users receive instant feedback when making mistakes and, when checking work after writing it, receive reports clearly indicating where issues were found, why they occurred, and how to correct them.

It is also highly customizable, allowing users to utilize the tool in the way that fits their needs best. Those who want to focus their reproducibility checking in a certain direction have that option and those who want widespread overviews can also have their needs meet. Users who are interested in going beyond the base functionality of `proj_analyze` and `proj_check` also have additional functions at their disposal that they can use to check reproducibility, file paths, file types, etc.

3.1   Potential Applications of `fertile`

These features make `fertile` an excellent tool for addressing the issue of scientific reproducibility on a widespread scale. `fertile` can provide a variety of benefits to users in all different application domains and with all different experiences. In this chapter, we consider the many potential uses of the package.

3.1.1   In Journal Review

As discussed in Chapter 1, Academic journals have a significant reproducibility problem. In an attempt to address this, many journals have instituted reproducibility policies for submitting researchers to follow. Although a variety of journals have these policies, particularly in the Statistical and Data Sciences, very few actually go through the process of verifying that the standards are met. Authors, finding it to to be a complicated and challenging task, will often not take the necessary steps to make their work truly reproducible. And journals, given the amount of time and money required to verify submissions' reproducibility, will often give submitting authors the benefit of the doubt in assuming that their work is reproducibile as long as some code and/or data has been provided. This results in the publishing of many articles that claim to be reproducible in theory, but do not meet such standards when tested in practice.

`fertile` could provide significant assistance with this process. Journals could integrate `fertile` into their article review workflow, ensuring that certain reproducibility checks were passed before an article could be accepted.

Depending on the level of detail with which the journal wanted to examine reproducibility, the integration could be done in a variety of ways. Here, we'll consider two:

1. Journal reviewers run `fertile` on every submitted R project.

Journals that desire a detailed summary of the reasons for reproducibility failure (such as information that one specific file was not commented enough)—and whose editors were willing to put in a little bit of time to collect this information—could choose to run `fertile` on all submitted papers that included R code.

They could require a list of checks to pass and provide a list of exceptions for cases in which checks could fail—for instance, a journal could state that even though they support good documentation, they do not require code to be fully commented.

Authors could run `fertile` on their work before submitting it to a journal to ensure that they passed the required list of checks and that any failures they saw were accounted for in the journal exceptions. Then journals, in order to ensure that authors followed the provided reproducibility guidelines, could run `fertile` on each submitted article as part of the review process. If the required checks were passed, the article could be accepted, but if they failed it would be rejected.

Although it would require some effort on the journal's side, this would still be an incredibly fast process: as long as a journal required that all submissions in `R` be in the `R Project` format, one reviewer could load the submission, run `fertile`, and receive a summary of the submission's reproducibility in a matter of minutes.
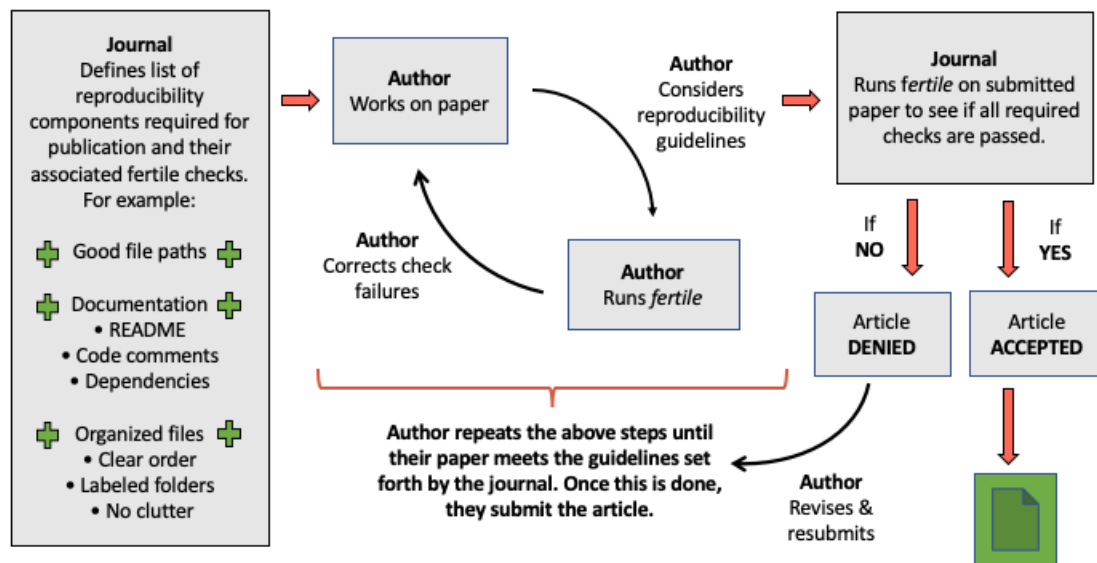
Figure 3.1: Potential 'fertile' Journal Review Process

2. Journal reviewers do not run `fertile` on each project, but instead require submitting authors to include a `fertile` summary sheet showing the reproducibility badges awarded to their work.

Some journals may desire a reproducibility summary, but not require details as specific as the exact reasons for failure. These journals could run a simplified approach to reproducibility review. Rather than spend the additional time to run `fertile` on every submission and review the reasons for check failure, the journals could instead require that authors include a coversheet—produced by the `proj_badges()` function—with their submission.

This cover sheet would show which of the six primary reproducibility components were met and which ones failed, a short summary of specific checks that were not

successful, and information about the cover sheet's generation: who generated it, when, and with which files.

Journals could place an acceptance requirement that articles achieve a certain subset of badges—for instance, file paths, randomness, and documentation. Submitting authors would run `fertile` on their end to see which badges they passed. Once they met the requirements, they could run `proj_badges()` to generate an article coversheet. This coversheet would then be submitted alongside the article and considered as part of the review process. All that reviewers would need to do to ensure that reproducibility requirements are met would be to look at the cover sheet to see which badges the project achieved and check the cover sheet generation information to ensure that it was truly produced by the project that the author says it was produced by.
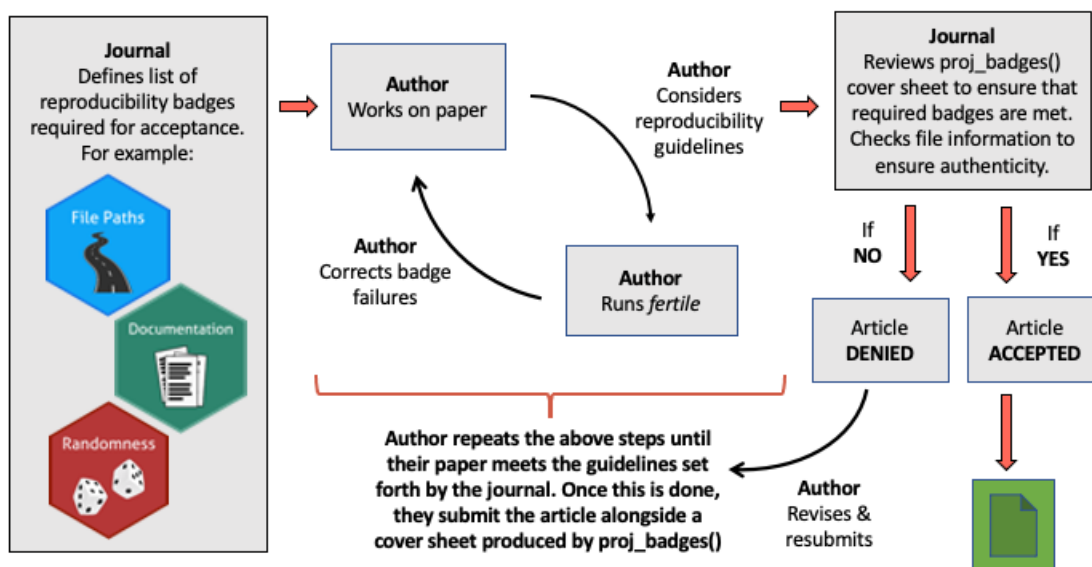
Figure 3.2: Another Potential 'fertile' Journal Review Process

Both of these processes would be much faster than that employed currently at the American Journal of Political Science, which goes through a thorough, multi-week-long reproducibility confirmation procedure for all submitted articles. Submitting researchers would know exactly which goals they were trying to achieve. They could download `fertile` on their own, run it on their project, check to see if their goals are met, and take the recommended steps to address failures if not. Then, upon submission, journals would only need to take minimal steps to ensure that those standards were met.

For journals that currently spend little effort on ensuring the reproducibility of submitted articles,`fertile` could provide significant benefit for minimal cost. Rather than spending money to hire a reproducibility editors, any journal board wanting to improve their publication's reproducibility could instead choose to implement `fertile`—for free—as part of the review workflow.

Journals that already have designated reproducibility editors, such as *Biostatistics* and the *Journal of the American Statistical Association*, could also see benefits from `fertile`(Biostatistics (2020), Journal of the American Statistical Association Editors (2020)). The integration of `fertile` could speed up the review process, allowing the editors to review more papers for reproducibility adherence in a shorter time.

Although this would only address a small aspect of reproducibility–that involving data analysis projects written in R–it would provide a significant time- and money-saving impact for both authors and reviewers in that domain.

3.1.2   For Teaching Reproducibility

`fertile` could also be integrated into Statistical and Data Sciences coursework in order to educate students on topics of reproducibility.

Many of the existing programs to teach reproducibility are courses focused on replication studies, where students must take a published paper and replicate the steps within completely. This process, which includes requesting the necessary data and code files from the original author(s) and sometimes even expanding the existing analysis further, often requires that participants have knowledge of data analysis and the scientific research process to be successful. As a result, such courses focused on reproducibility tend to exist only at the graduate level.

5   Replication Paper Assignment

For graduate students enrolled in Gov2001, the main class assignment is to write a re-search paper that replicates and extends an existing scholarly work, while applying some advanced method to a substantive problem in some substantive field of study. Most un-dergraduate students enrolled in Gov1002 and all non-Harvard students enrolled in Stat E-200 complete a final exam instead of the replication paper (more information for these students in Section 10).

Your goal for this assignment is to produce a paper that is publishable in a scholarly journal — something we assume you have never done before and do not presently have any idea how to do. We will show you! Detailed information on the assignment can be found in an article I wrote about it called "Publication, Publication," at `GaryKing/ papers` along with continuing updates. For initial versions of some papers from recent years, see the class Dataverse at `j.mp/G2001dv`.

As this assignment involves carefully choreographed hand-offs and interactions that connects you to everyone else in class, please respect these deadlines. Everyone is de-pending on you. Here's the list of requirements; the exact date and time of the deadline for each is given on the class web site, `j.mp/G2001`.

1. Identify your coauthors. All papers must be coauthored with one or two other mem-bers of this class. If you have problems, or would like suggestions, talk to the TFs.

2. Identify a scholarly article to replicate that meets our specific criteria. Upload to Canvas a PDF copy of the article, along with a brief paragraph (of less than about 200 words) explaining your choice. This paragraph must also list a classmate (out-side your group) willing to testify that your article choice met all the criteria listed in "Publication, Publication".

3. Turn in a draft of your paper with completed figures and tables, and a proposed outline of the paper, in a relatively polished form. This draft need not have much text yet (although the more you complete, the more useful comments you will get in return). Also turn in a *replication data file*, with all of the data and information necessary to replicate your results and reproduce your tables and figures. At the same time, we will assign your paper to several other students to replicate, and you will receive another group's paper to replicate.

4. Replicate the other group's proto-paper and write a memo to them (with a copy to us), pointing out ways to improve their paper and analysis. You will be evaluated based on how helpful, not how destructive, you are. The best comments are written so fellow students can hear and learn from them rather than trying to demonstrate how smart you are.

5. Turn in the final version of the paper. By the same deadline, you must also follow standard academic practice and create a permanent replication data archive by up-loading all your data and code to the Gov2001 Dataverse (`j.mp/G2001dv`). If you would like an *extension* with this (and only this) deadline, you do not need to ask permission: We will accept papers for a week after the due date given on the class web site (although since you will have had more time, papers turned in after the original deadline will be graded according to proportionately higher standards).

6

Figure 3.3:   The Replication Assignment From Harvard Pro-fessor Gary King's Gov 2001 Graduate Course (Source: https://projects.iq.harvard.edu/files/gov2001/files/syllabus.pdf)

Undergraduate students, therefore, do not get exposed to reproducibility very often.

There are a few exceptions–for instance, introductory courses at Smith College and Duke University that integrate RMarkdown to promote reproducible workflows–but overall, reproducibility is not covered in data science courses the undergraduate level.

`fertile` could help change this, allowing for many more colleges and universities to integrate reproducibility into their courses. The barriers to entry for using and benefitting from the package are very low, requiring only that participating students have:

- R and RStudio installed on their computer
- Knowledge of how to install a package from GitHub and load it into their environment
- Knowledge of how to create an R project
- Knowledge of how to run basic functions and input simple file paths

Though the process may entail some confusion and troubleshooting at first, even those brand new to `R` could succeed in overcoming these barriers in only a few days of class. As a result, `fertile` could provide professors with an opportunity to teach reproducibility concepts in introductory level courses.

`fertile` could easily integrate into coursework in a similar way to how RMarkdown was integrated at Smith College and Duke University. While there is not only one way to utilize the software in class, a potential use of `fertile` could look as follows:

At the beginning of their courses, the professor provides their students with a brief introduction to reproducibility, including its importance and a basic description of how it is achieved. Shortly after, they introduce R Projects and the `fertile` package, explaining that they are tools to help with reproducibility. Then, they institute a requirement for all submitted homework assignments in the course: students must create and submit their work in the R Project format, but prior to submission must run `fertile` on their project to ensure that it passes reproducibility standards. When reproducibility errors inevitably occur, they can be used as teaching moments: the professor can share the error, explain why it happened, walk through `fertile`'s response to it, and interactively work with students to illustrate how it can be fixed.

The integration of `fertile` in this way would be an excellent method to introduce students to reproducibility concepts early on in their data science education, but at a low cost to the professor.

There are a variety of benefits to introducing students to reproducibility sooner, rather than later–in graduate school or through independent research on the topic:

- Teaching reproducibility early on gives students important research tools and understanding before they conduct any of their own important analysis.

- Practicing before students are believed to be skilled and highly educated in data science gives them an opportunity to fail and learn without fear of judgement.

- Integrating concepts early helps ingrain them in the minds of students, ensuring that reproducibility begins to come naturally to them.

These students would then be prepared for entering the research world and contributing to data science work in a transparent and reproducible way.

** INTRODUCE EXPERIMENT HERE **

`fertile` is designed to: 1) be simple enough that users with minimal `R` experience can use the package without issue, 2) increase the reproducibility of work produced by its users, and 3) educate its users on why their work is or is not reproducible and provide guidance on how to address any problems.

To test `fertile`'s effectiveness, we began an initial randomized control trial of the package on an introductory undergraduate data science course at Smith College in Spring 2020 **ADD FOOTNOTE** (This study was approved by Smith College IRB, Protocol #19-032).

The experiment was structured as follows:

1.Students are given a form at the start of the semester asking whether they consent to participate in a study on data science education. In order to successfully consent, they must provide their system username, collected through the command `Sys.getenv("LOGNAME")`. To maintain privacy the results are then transformed into a hexadecimal string via the `md5()` hashing function.

2. These hexadecimal strings are then randomly assigned into equally sized groups, one experimental group that receives the features of `fertile` and one group that receives a control.

3. The students are then asked to download a package called `sds192` (the course number and prefix), which was created for the purpose of this trial. It leverages an `.onAttach()` function to scan the `R` environment and collect the username of the user who is loading the package and run it through the same hashing algorithm as used previously. It then identifies whether that user belongs to the experimental or the control group. Depending on the group they are in, they receive a different version of the package. The structure of this function can be seen below:

```r
# .onAttach() from the sds192 package
# Run automatically any time the sds192 package is loaded

.onAttach <- function(libname, pkgname) {

  # The experimental group gets `fertile` loaded secretly
  if (is_experimental()) {
    suppressMessages(library(fertile))
  }
}

is_experimental <- function(logname = whoami::username()) {

  # Students are placed into experimental and control groups but remain anonymous
  fertile_group <- c(
    "f7b0a9d5117b88cecec122f8ba0e52fb", "4d0295a810fb8491f91f914771572485",
```

```
    "36211a1f19f82ae07aed990b671c9b20", "b5d2b72b4f36f3afdce32a8409dc6ea0",
    "d498227fd9e6a4c42494bbebc42f6aa8", "191922566ef6a9910682ad9836b6d018",
    "b1b8278d6b7eecd2c595ab6138de17e0", "4894c3d932fa6cfc9ec59214c41f12c7",
    "2f1e21316352049069f8e4351d6cb88d", "73f4a06a26ad5342e30e2e7fdf92dba2",
    "9ff232c0e56f04be4ba2037aec6471b2", "abb2b711c8d45ffac12006945e10ed31",
    "3a49dd3c6e591933a0909734a94ca37c", "db1b731709316be69dae1d56382e4243",
    "8e76b368a5e999869c7ca8f9f1566cf8"
  )
  control_group <- c(
    "9aa36583f54766205850428e8f1a4c89", "fdc61d4d4c93473be5f485eea55140b3",
    "f03020938b31818063c79d2422755183", "d0a981421d0e378e26186f99c618c748",
    "7ec57b1f2bca9ac1e702fb68427b781b", "8857ca4d5e1fe92ae70b1ba95c0b7b8b",
    "e5e30623e9d09d29ded851b7fb40cb51", "592572bb9fce168f37117fd0d6e0e5ee",
    "c40b1786558a15f8dd151da163ebd0b4", "67b8da3952e07ba3f2c5c715c4042220",
    "f5ca3188ff14cbf43b6d4cd5b225376e", "b3b222f42beaae6ca2b309b94b1340be",
    "2dbac7345a71a13c434af882853e86bd", "3c5df10c572b5627418c034f65f52dee",
    "0ff07a4f6faaf3df57b348dad7bd22c9"
  )

  digest::digest(logname, algo = "md5") %in% fertile_group

}
```

4. The experimental group receives the basic `sds192` package, which consists of some data sets and R Markdown templates necessary for completing homework assignments and projects in the class, but also has `fertile` installed and loaded silently in the background. The package's proactive features are enabled, and therefore users will receive warning messages when they use absolute or non-portable paths or attempt to change their working directory. The control group receives only the basic `sds192` package, including its data sets and R Markdown templates. All students from both groups then use their version of the package throughout the semester on a variety of projects.

5. Both groups are given a short quiz on different components of reproducibility that are intended to be taught by `fertile` at both the beginning and end of the semester. Their scores are then compared to see whether one group learned more than the other group or whether their scores were essentially equivalent. Additionally, for every homework assignment submitted, the professor takes note of whether or not the project compiles successfully.

Based on the results, we hope to determine whether `fertile` was successful at achieving its intended goals. A lack of notable difference between the *experimental* and *control* groups in terms of the number of code-related questions asked throughout the semester would indicate that `fertile` achieved its goal of simplicity. A higher average for the *experimental* group in terms of the number of homework assignments

that compiled successfully would indicate that `fertile` was successful in increasing reproducibility. A greater increase over the semester in the reproducibility quiz scores for students in the *experimental* group compared with the *control* group would indicate that `fertile` achieved its goal of educating users on reproducibility. Success according to these metrics would provide evidence showing `fertile`'s benefit as tool to help educators introduce reproducibility concepts in the classroom.

3.1.3   In Other Areas

`fertile` can also provide benefits in other domains. Though not an exhausted list, some of the potential uses of the software are:

- *Private Companies*: Data analysis-focused companies could require their employees to use `fertile` to check the reproducibility of their projects before presenting them to clients. This would help such companies ensure that clients could trust the results that were being produced.

- *Conferences*: Similar to academic journals, conferences promoting open research could require that papers written in `R` pass a `fertile` check as a condition for acceptance. Even if there were an exception given for those using confidential/identifiable data, this would likely increase the overall reproducibility of conference papers significantly.

- *Informal Data Analysis*: A lot of content in the `R` community is created purely for fun and interest. Outside of work, many `R` users will create data visualizations or analyses for their own private blogs or their twitter. Sometimes, users will also participate in community events like Tidy Tuesday, a weekly social project where a data set is posted and users are asked to analyze it and create a visualization of their choice. Many people use these informal analyses as an opportunity for learning and discussion, often sharing them on social media to try and get feedback on their work. Ensuring that the work is reproducible would facilitate this process. Users could run their project files through `fertile` to check that they are reproducible and post a link to download them. This would then allow others to run the code on their own to understand how it works and more easily be able to make suggestions as to how to improve it!

`fertile` is incredibly versatile in its applicability. It can be used anywhere from informal data analysis projects to academic journal review.

Potential sources:

https://arxiv.org/abs/1401.3269

https://academic.oup.com/isp/article-abstract/17/4/392/2528285

https://berkeleysciencereview.com/2014/06/reproducible-collaborative-data-science/

https://guides.lib.uw.edu/research/reproducibility/teaching

https://escholarship.org/uc/item/90b2f5xh

Conclusion

In the field of data science, research is considered fully *reproducible* when the requisite code and data files produce identical results when run by another analyst. Though there is no clear consensus on the exact standards necessary to achieve reproducibility, writing on the topic identifies several components of great importance:

1. The basic project components are made accessible to the public.

2. The file structure of project is well-organized.

3. The project is documented well.

4. File paths used in code are not system- or user-dependent.

5. Randomness is accounted for.

6. Code is readable and consistently styled.

Reproducibility is incredibly important to the scientific community, helping ensure the accuracy of the findings of studies and data analyses and simplifying the process of collaboration and knowledge sharing. When all of the files necessary for a study to be run are published simultaneously, it makes it much easier for others to understand the methods and ideas that were used and apply them to other work in similar areas, promoting the process of scientific advancement.

However, even though it has many benefits in the scientific community, reproducibility is currently facing a crisis. The vast majority of researchers across all scientific fields have been unable to reproduce another researcher's results, and many have been unable to reproduce even their own. In some fields, more than half of published articles have failed attempts at reproducibility.

Researchers across the sciences have recognized this problem and taken steps to address it. Data-intensive academic journals have put in place reproducibility guidelines requiring that submitted articles meet at least some of the standards listed above. Software developers have built tools to help data analysts make their projects more reproducible. These include a small library of R packages, which provide benefits to users of the popular statistical language R, as well as several continuous integration tools, which are much more broad in their application. Educators have also taken action on reproducibility, introducing courses and workshops focused on the topic at their universities.

However, many of these solutions have fallen short. Even with reproducibility guidelines, many journals still publish work that is not reproducible. Journal review takes significantly longer and requires many more resources when ensuring that reproducibility guidelines are fully met. Journals often cannot spare the additional time and money, leaving it up to submitting authors to ensure that their articles meet standards. Since the adherence to standards is not often checked, and achieving full reproducibility can sometimes be time consuming and challenging, many authors fail to take this step.

Software solutions have not fared much better. Many of the `R` specific solutions only address a single component of reproducibility—for example, version control—and leave out the rest, making it challenging for researchers to address their problems all at once. Additionally, a lot of the software is quite complex in its operation, containing barriers to entry that prevent less-experienced `R` users from trying them. The continuous integration tools do not face the same issues, but have their own problems. Due to their general nature, these tools do not have any way of addressing coding-language-specific reproducibility problems such as managing software dependencies.

Finally, educational attempts to address reproducibility are inaccessible and time consuming to implement. Many of the existing educational options require a great deal of research knowledge, and as a result are only available at the graduate level, leaving undergraduate students with little to no exposure on the subject. Additionally, many current versions of reproducibility education require a lot of time and focus in class. Professors wanting to share the topic of reproducibility with their students must do so while sacrificing a significant portion of time from other important topics.

Due to their many associated shortcomings, none of these solutions appear to have the potential to address the problem of reproducibility on a wider scale. That is where my work comes in. I have worked to develop `fertile`, a package built for `R` users that is focused on creating optimum conditions for reproducibility for data analysis projects written in `R`.

The package is designed to possess none of the challenges apparent in other solutions—to serve as a one-stop solution, where users can go to address all of their reproducibility needs at one time. Rather than being time consuming, complicated, inaccessible, or too focused on one issue, `fertile` is:

1) Simple, with a small library of functions/tools that are straightforward to use. Some of the package's functionalities, like the interactive file path warning system, require no effort from the user to enable, while most others can be run with only a handful of functions requiring one argument: a path to a project directory.

2) *Accessible to a variety of users, with a relatively small learning curve.* Users do not need a background in research, nor do they need an advanced knowledge of `R` programming, to run the majority of functions in `fertile`.

3) *Able to address a wide variety of aspects of reproducibility, rather than just one or two key issues.* `fertile` contains functions to address all six previously

defined reproducibility components, unlike most software which addresses only one or two.

4) *Language specific, possessing features that address some of the reproducibility challenges associated with* `R`. The `has_no_randomness()`, `proj_pkg_script()`, and `proj_dependency_report()` functions handle the `R`-specific issues of random number generation and package dependencies.

5) *Customizable, allowing users to choose for themselves which aspects of reproducibility they want to focus on.* There is no one way that users are forced to use `fertile`. Those who only want to focus on one aspect of reproducibility can do so, while those who want to address everything at once have that option as well.

6) *Educational, teaching those that use it about why their projects are not reproducible and how to correct that in the future.* Throughout the reproducibility checking process, `fertile` provides informative messages that explain to users the flaws in their code and show them ways to correct the issues.

Due to its many advantages compared with traditional reproducibility solutions, `fertile` has the potential to provide significant benefits in a variety of domains, including in the areas of journal review and education, where other solutions have not quite met the mark.

In the journal review process, `fertile` can be integrated to help ensure that submitted articles meet reproducibility standards. Since `fertile` is both free and publicly available, authors and journal reviewers could take advantage of it in their reproducibility-checking process. Journals might require that certain `fertile` checks be passed for articles to be approved, submitting authors could ensure that their projects pass on their end, and then all that would be required for journals to ensure reproducibility would be a quick run of `fertile`'s functions. This would greatly reduce the barriers on both the reviewing and authoring side of the issue and would vastly speed up the reproducibility-checking process, making it much easier to ensure that published articles are truly reproducible.

In the classroom, `fertile` could be used to extend reproducibility education to undergraduate students—even those at the introductory level. Due to its simplicity, the package would take minimal time and effort to integrate into courses, allowing professors to teach reproducibility without taking important time away from other key topics. If professors were to integrate `fertile` in to their courses, students would be able begin learning about reproducibility much earlier in their data science careers than would likely happen otherwise, increasing the chance that they prioritize reproducibility in their future work.

Academia and research are not the only applications, however. `fertile` could also assist the general world of data science. `R` bloggers and Tidy Tuesday participants could use the package to improve the quality of their code- and data- sharing, making it easier for others to learn from their work. Data journalists, in some situations, could do something similar, sharing the (reproducible) code and data behind their projects to remove the black-box nature that some such projects possess, building

a relationship of trust with their readers. And private companies doing consulting work could integrate `fertile` into their workflow as a way of increasing transparency between themselves and their clients.

Appendix A

The First Appendix

This first appendix includes all of the R chunks of code that were hidden throughout the document (using the `include = FALSE` chunk tag) to help with readability and/or setup.

In the main Rmd file

```r
# This chunk ensures that the thesisdown package is
# installed and loaded. This thesisdown package includes
# the template files for the thesis.
if (!require(remotes)) {
  if (params$`Install needed packages for {thesisdown}`) {
    install.packages("remotes", repos = "https://cran.rstudio.com")
  } else {
    stop(
      paste('You need to run install.packages("remotes")',
            "first in the Console.')
    )
  }
}
if (!require(thesisdown)) {
  if (params$`Install needed packages for {thesisdown}`) {
    remotes::install_github("ismayc/thesisdown")
  } else {
    stop(
      paste(
        "You need to run",
        'remotes::install_github("ismayc/thesisdown")',
        "first in the Console."
      )
    )
  }
}
library(thesisdown)
```

```r
# Set how wide the R output will go
```

Appendix B

The Second Appendix, for Fun

References

American Economic Association. (2020). Data and code availability policy. Retrieved from `https://www.aeaweb.org/journals/data/data-code-policy`

American Journal of Political Science. (2016, May). Guidelines for preparing replication files. Retrieved from `https://ajps.org/wp-content/uploads/2018/05/ajps_replication-guidelines-2-1.pdf`

American Statistical Association. (2020). JASA acs reproducibility guide. Retrieved from `https://jasa-acs.github.io/repro-guide/pages/author-guidelines`

Baker, M. (2015). Over half of psychological studies fail reproducibility test. *Nature.* Retrieved from `https://www.nature.com/news/over-half-of-psychology-studies-fail-reproducibility-test-1.18248`

Baker, M. (2016). 1,500 scientists lift the lid on reproducibility. *Nature.* Retrieved from `https://www.nature.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970`

Baumer, B., Cetinkaya-Rundel, M., Bray, A., Loi, L., & Horton, N. J. (2014). R markdown: Integrating a reproducible analysis tool into introductory statistics. *arXiv Preprint arXiv:1402.1894.*

Begley, C. G., & Ellis, L. M. (2012). Raise standards for preclinical cancer research. *Nature, 483*(7391), 531–533.

Biostatistics. (2020). Information for authors. Retrieved from `https://academic.oup.com/biostatistics/pages/General_Instructions`

Blischak, J., Carbonetto, P., & Stephens, M. (2019). Workflowr: A framework for reproducible and collaborative data science. Retrieved from `https://CRAN.R-project.org/package=workflowr`

Bollen, K., Cacioppo, J. T., Kaplan, R. M., Krosnick, J. A., Olds, J. L., & Dean, H. (2015). Report of the subcommittee on replicability in science advisory committee to the nsf sbe directorate. Retrieved from `https://www.nsf.gov/sbe/SBE_Spring_2015_AC_Meeting_Presentations/Bollen_Report_on_Replicability_SubcommitteeMay_2015.pdf`

Broman, K. (2019). Initial steps toward reproducible research: Organize your data and

code. *Sitewide ATOM*. Retrieved from `https://kbroman.org/steps2rr/pages/organize.html`

Cambridge University Press. (2020). Experimental results - transparency and openness policy. Retrieved from `https://www.cambridge.org/core/journals/experimental-results/information/transparency-and-openness-policy`

Claerbout, J. F., & Karrenbach, M. (1992). Electronic documents give reproducible research a new meaning. In *SEG technical program expanded abstracts 1992* (pp. 601–604). Society of Exploration Geophysicists.

Cooper, N., Hsing, P.-Y., Croucher, M., Graham, L., James, T., Krystalli, A., & Michonneau, F. (2017). A guide to reproducible code in ecology and evolution. *British Ecological Society*. Retrieved from `https://www.britishecologicalsociety.org/wp-content/uploads/2017/12/guide-to-reproducible-code.pdf`

Eisner, D. A. (2018). Reproducibility of science: Fraud, impact factors and carelessness. *Journal of Molecular and Cellular Cardiology*, *114*, 364–368. `http://doi.org/https://doi.org/10.1016/j.yjmcc.2017.10.009`

Epstein, M. L., Lazarus, A. D., Calvano, T. B., Matthews, K. A., Hendel, R. A., Epstein, B. B., & Brosvic, G. M. (2002). Immediate feedback assessment technique promotes learning and corrects inaccurate first responses. *The Psychological Record*, *52*(2), 187–201.

Fidler, F., & Wilcox, J. (2018). Reproducibility of scientific results. In E. N. Zalta (Ed.), *The stanford encyclopedia of philosophy* (Winter 2018). `https://plato.stanford.edu/archives/win2018/entries/scientific-reproducibility/`; Metaphysics Research Lab, Stanford University.

FitzJohn, R., Ashton, R., Hill, A., Eden, M., Hinsley, W., Russell, E., & Thompson, J. (2020). Orderly: Lightweight reproducible reporting. Retrieved from `https://CRAN.R-project.org/package=orderly`

Gancarz, M. (2003). *Linux and the unix philosophy* (2nd ed.). Woburn, MA: Digital Press.

Goodman, S. N., Fanelli, D., & Ioannidis, J. P. A. (2016). What does research reproducibility mean? *Science Translational Medicine*, *8*(341), 1–6. `http://doi.org/10.1126/scitranslmed.aaf5027`

Gosselin, R.-D. (2020). Statistical analysis must improve to address the reproducibility crisis: The access to transparent statistics (acts) call to action. *BioEssays*, *42*(1), 1900189. `http://doi.org/10.1002/bies.201900189`

Hardwicke, T. E., Mathur, M. B., MacDonald, K., Nilsonne, G., Banks, G. C., Kidwell, M. C., . . . others. (2018). Data availability, reusability, and analytic reproducibility: Evaluating the impact of a mandatory open data policy at the journal cognition. *Royal Society Open Science*, *5*(8), 180448. Retrieved from

https://royalsocietypublishing.org/doi/full/10.1098/rsos.180448

Henry, L., & Wickham, H. (2020). Tidyselect: Select from a set of strings. Retrieved from `https://CRAN.R-project.org/package=tidyselect`

Hermans, F., & Aldewereld, M. (2017). Programming is writing is programming. In *Companion to the first international conference on the art, science and engineering of programming* (pp. 1–8).

Hillenbrand, S. (2014). Reproducible and collaborative: Teaching the data science life. Berkeley Science Review. Retrieved from `https://berkeleysciencereview.com/2014/06/reproducible-collaborative-data-science/`

Horton, N. J., Baumer, B. S., & Wickham, H. (2014). Teaching precursors to data science in introductory and second courses in statistics. *arXiv Preprint arXiv:1401.3269.*

Hrynaszkiewicz, I. (2020). Publishers' responsibilities in promoting data quality and reproducibility. *Handbook of Experimental Pharmacology, 257*, 319–348. `http://doi.org/https://doi.org/10.1007/164_2019_290`

Jacoby, W. G., Lafferty-Hess, S., & Christian, T.-M. (2017). Should journals be responsible for reproducibility? Inside Higher Ed. Retrieved from `https://www.insidehighered.com/blogs/rethinking-research/should-journals-be-responsible-reproducibility`

Janz, N. (2016). Bringing the gold standard into the classroom: Replication in university teaching. *International Studies Perspectives, 17*(4), 392–407.

Journal of Computational and Graphical Statistics. (2020). Instructions for authors. Retrieved from `https://www.tandfonline.com/action/authorSubmission?show=instructions&journalCode=ucgs20`

Journal of Statistical Software. (2020). Instructions for authors. Retrieved from `https://www.jstatsoft.org/pages/view/authors#review-process.`

Journal of the American Statistical Association Editors. (2020). JASA editors talk reproducibility. Retrieved from `https://www.amstat.org/ASA/Publications/Q-and-As/JASA-Editors-Talk-Reproducibility.aspx`

Kitzes, J., Turek, D., & Deniz, F. (2017). *The practice of reproducible research: Case studies and lessons from the data-intensive sciences.* Berkeley, CA: University of California Press. Retrieved from `https://www.practicereproducibleresearch.org`

Leopold, S. S. (2015). Editorial: Increased manuscript submissions prompt journals to make hard choices. *Clinical Orthopaedics and Related Research, 473*(3), 753–755. `http://doi.org/10.1007/s11999-014-4129-1`

Lupia, A., & Elman, C. (2014). Openness in political science: Data access and research

transparency. *PS, Political Science & Politics*, *47*(1), 19.

Martinez, C., Hollister, J., Marwick, B., Szöcs, E., Zeitlin, S., Kinoshita, B. P., . . .
    Meinke, B. (2018). Reproducibility in Science: A Guide to enhancing reproducibility
    in scientific results and writing. Retrieved from `http://ropensci.github.io/`
    `reproducibility-guide/`

Marwick, B. (2019). Rrtools: Creates a reproducible research compendium. Retrieved
    from `https://github.com/benmarwick/rrtools`

Marwick, B., Boettiger, C., & Mullen, L. (2018). Packaging data analytical work
    reproducibly using R (and friends). *The American Statistician*, *72*(1), 80–88.
    `http://doi.org/doi.org/10.1080/00031305.2017.1375986`

McArthur, S. L. (2019). Repeatability, reproducibility, and replicability: Tackling the
    3R challenge in biointerface science and engineering. *Biointerphases*, *14*(2), 1–2.
    `http://doi.org/10.1116/1.5093621`

McIntire, E. J. B., & Chubaty, A. M. (2020). Reproducible: A set of tools that enhance
    reproducibility beyond package management. Retrieved from `https://CRAN.R-`
    `project.org/package=reproducible`

National Institutes of Health. (2014, June). Principles and guidelines for re-
    porting preclinical research. Retrieved from `https://www.nih.gov/research-`
    `training/rigor-reproducibility/principles-guidelines-reporting-`
    `preclinical-research`

OpenSci, R. (2020). Drake: A pipeline toolkit for reproducible computation at scale.
    Retrieved from `https://cran.r-project.org/package=drake`

Oracle Corporation. (2019). Wercker. Retrieved from `https://github.com/`
    `wercker/wercker`

R Journal Editors. (2020). Instructions for authors. Retrieved from `https://`
    `journal.r-project.org/share/author-guide.pdf`

R-Core-Team. (2020). Writing r extensions. *R Foundation for Statistical Computing*.
    Retrieved from `http://cran.stat.unipd.it/doc/manuals/r-release/R-exts.`
    `pdf`

Ross, N., DeCicco, L., & Randhawa, N. (2018). Checkers: Automated checking
    of best practices for research compendia. Retrieved from `https://github.com/`
    `ropenscilabs/checkers/blob/master/DESCRIPTIONr`

Stodden, V., Seiler, J., & Ma, Z. (2018a). An empirical analysis of journal policy effec-
    tiveness for computational reproducibility. *Proceedings of the National Academy of
    Sciences*, *115*(11), 2584–2589. Retrieved from `https://www.pnas.org/content/`
    `115/11/2584`

Stodden, V., Seiler, J., & Ma, Z. (2018b). An empirical analysis of journal policy

effectiveness for computational reproducibility. *Proceedings of the National Academy of Sciences*, *115*(11), 2584–2589. `http://doi.org/10.1073/pnas.1708290115`

The American Statistician. (2020). Instructions for authors. Retrieved from `https://www.tandfonline.com/action/authorSubmission?show=instructions&journalCode=utas20`

Ushey, K., & RStudio. (2020). Renv: Project environments. Retrieved from `https://cran.r-project.org/web/packages/renv/index.html`

Wallach, J. D., Boyack, K. W., & Ioannidis, J. P. A. (2018). Reproducible research practices, transparency, and open access data in the biomedical literature, 2015-2017. *PLOS Biology*, *16*(11), 1–20. `http://doi.org/10.1371/journal.pbio.2006930`

Wickham, H. (2015). *R packages* (1st ed.). Sebastopol, CA: O'Reilly Media, Inc.

Wilson, G., Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., . . . others. (2014). Best practices for scientific computing. *PLOS Biology*, *12*(1), e1001745.

Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., & Teal, T. K. (2017). Good enough practices in scientific computing. *PLOS Biology*, *13*(6), e1005510. Retrieved from `https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005510`

Woolston, C. (2020). TOP factor rates journals on transparency, openness. Nature Index. Retrieved from `https://www.natureindex.com/news-blog/top-factor-rates-journals-on-transparency-openness`

Yu, B., & Hu, X. (2019). Toward training and assessing reproducible data analysis in data science education. *Data Intelligence*, *1*(4), 381–392.