Addressing The Scientific Reproducibility Crisis Through Educational Software Integration

Audrey M. Bertin

Submitted to the Program in Statistical and Data Sciences
of Smith College
in partial fulfillment
of the requirements for the degree of
Bachelor of Arts

Benjamin S. Baumer, Primary Faculty Advisor
Albert Y. Kim, Secondary Faculty Advisor

May 2021

Acknowledgements

This project would not have been possible without the guidance of Professor Ben Baumer, who helped inspire my interest in data science and mentored me throughout this project for the last two years. I would also like to extend my thanks to Jenny Bryan and Hadley Wickham, whose advice has helped guide the development of several features in `fertile`, along with the authors of all of the R packages utilized by `fertile` to achieve its functionality. Additionally, a huge thanks to the many introductory data science students who volunteered their time as part of a study to test the effectiveness of implementing `fertile` in the classroom. The help of all these Smithies was invaluable in ensuring that the overall `fertile` product was as user friendly and effective as possible.

Table of Contents

List of Figures

Abstract

Data science research is considered *reproducible* when the associated code and data files produce identical results when run by another analyst. Although reproducibility is a key component in the advancement of scientific knowledge, a significant proportion of research articles and other analyses fail to meet reproducibility standards. Steps have been taken to address this issue, including academic courses on reproducibility, additional requirements or recommendations for journal article acceptance, and a variety of software tools. However, many of these are challenging to use, are too generalized, or are not accessible to a wide audience. In this thesis, I present my work on developing `fertile`, an `R` package designed to help improve the reproducibility of `R Projects` and address the limitations of other solutions by being 1) simple to use, 2) easily accessible, 2) broad in scope, 3) tailored to the specific challenges faced by R users, 4) customizable, and 5) educational. Chapter 1 considers the background information motivating `fertile`, including explanation of reproducibility, its issues, current solutions, and their limitations. Chapter 2 is code-focused, demonstrating the functions available in `fertile` to address different aspects of reproducibility and delving into some of the details of how the software works. Finally, Chapter 3 considers `fertile`'s potential applications in the real world, including an in-depth analysis of an experiment involving `fertile`'s integration into an introductory data science course at Smith College.

Prologue

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 adviser—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 (Baumer, Cetinkaya-Rundel, Bray, Loi, & Horton (2014)), 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 exist now were there at the time, but there were not many features beyond the basics. Additionally, several functions were not fully written or were

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 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 definitely 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 (see `https://stackoverflow.com/questions/43638773/comprehensive-way-to-check-for-functions-that-use-the-random-number-generator-in`).

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?

Research in the field of data science is considered fully *reproducible* when the relevant 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)).

There are many benefits to reproducibility in scientific research. When researchers share their work in a reproducible format, readers can much more easily determine the accuracy of any findings by following the steps from raw data to conclusions. The creators of reproducible work can also more easily receive more specific feedback on their work, by allowing others to look through their process and even attempt pieces of the work themselves. Moreover, reproducibility makes it much simpler to pass on knowledge. Others interested in the same research topic as in a published work can review the associated code and/or data and then apply some of the the methods and ideas to their own work.

Although often confused, the concept of *reproducibility* is distinct from the related 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 reproducibility; the findings of research studies can not be accepted unless a variety of other researchers come to the same conclusions through independent work.

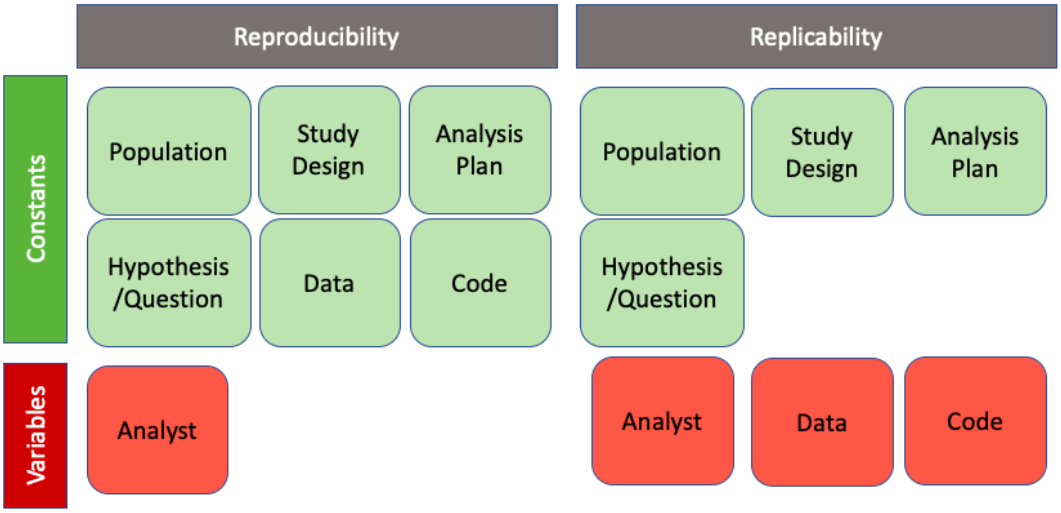


Figure 1.1: Reproducibility vs. Replicability — In reproducibility, data and code remain constant, while in replicability they are variable.

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 robust hypotheses.

1.2 The Reproducibility Crisis

Despite the apparent 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: Baker (2015) found that over 50% of studies psychology failed reproducibility tests and Begley & Ellis (2012) found that figure closer to 90% in the field of cancer biology.

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 newfound 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 significantly in the last several years. For example:

1. Eisner (2018) writes that much of science cannot be reproduced, arguing that some of the major factors behind the problem are fraud, encouraged by the need to publish in top journals, poor experimental design, and poor statistical analysis.

2. Fidler & Wilcox (2018) argue that the term "reproducibility crisis" has gained significant currency over the last decade after many large-scale reproducibility studies—including the *Nature* study discussed previously—have returned negative results.

3. Gosselin (2020) shares a call to action (summarized by the acronym ACTS—**AC**cess to **T**ransparent **S**tatistics), assembling measures that can be implemented by journals to increase the quality of statistical publications: standardizing paragraph contents, requiring a paragraph on statistical limitations, providing funding to study reproducibility, and requiring that methods sections start with statistics.

4. McArthur (2019) argues that a common theme in the issue of reproducibility and related challenges is the "increasing demands of complex research requiring use of multiple experimental and computational research methods" and promises to take steps to improve reproducibility in the *Biointerphase* journal.

5. Wallach, Boyack, & Ioannidis (2018) survey a random sample of biomedical journal articles in order to analyze their data availability for the purposes of reproducibility. They find results that are somewhat more positive than previous studies—indicating that perhaps the biomedical sciences are moving in the right direction—but argue that there is still a long way to go before true journal reproducibility is achieved.

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. "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 question can be found across a variety of fields. However, as Goodman, Fanelli, & Ioannidis (2016) argue, both the framework and terms used to describe reproducibility vary significantly across the sciences, and the scientific community has been unable to agree upon universally applicable clear standards.

At a minimum, according to Goodman, Fanelli, & Ioannidis (2016), achieving reproducibility requires the sharing of data (either raw or processed), relevant metadata, code, and related software. However, according to others, the full achievement of reproducibility may require additional components.

Kitzes, Turek, & Deniz (2017) present a series of case studies on reproducibility practices from across the data-intensive sciences, illustrating many different 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 analysis that extend beyond the minimum recommendations of Goodman, Fanelli, & Ioannidis (2016):

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

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

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 files, and include a descriptive `README` file. 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.

R OpenSci, a non-profit initiative founded in 2011 to make scientific data retrieval reproducible, has shared recommendations which focus on similar principles to those discussed previously. They emphasize the need for a well-developed file system, with clear labeling and no extraneous files. They also reiterate the necessity of noting dependencies and using 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:

Component	Requirements to Achieve Success in Component
1. The basic components of project are made accessible to the public	• *Data (raw and/or processed)* • *Metadata* • *Code* • *Related software*
2. The file structure is well-organized	• *Separate folders for different file types* • *No unnecessary files* • *Minimal clutter*
3. The project is well-documented	• *Files clearly named (preferably in way where run order is clear)* • *README is present* • *Code contains comments* • *Software dependencies are noted*
4. The file paths used in code are not system- or user-dependent	• *No absolute paths* • *No paths leading to locations outside the project directory* • *Only relative paths to locations within the directory*
5. Randomness is accounted for	• *If there is randomness in the code, a seed must be set*
6. The code is readable and consistently styled	• *Code must be written in a coherent, consistent, easy-to-read style* *Note: this component was not mentioned in literature, but it is included due to its importance for improving project readability (Hermans & Aldewereld (2017)).*

Figure 1.2: The 6 Major Components of Reproducibility — Accessibility, organization, documentation, good file paths, controlled randomness, style/readability.

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 reproducing 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. Note that this requirement goes beyond most of what is recommended in reproducibility literature, which simply recommends code to produce the analysis results—and not necessarily all outputs.
- 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 representing 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, Fanelli, & Ioannidis (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. No information about dependencies or software version needs to be included. Beyond that, there is not even a requirement that analysis be completed in a scriptable programming language—allowing for the potential that some submitted work may be entirely non-reproducible due to its format.

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 underpin 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.

In this section, we'll consider in detail the reproducibility policies of some of these top journals, selected by a compound measure of influence, impact, and prestige—which summarizes the average number of weighted citations received in a given year compared with the number of documents published in that journal over the previous three years—calculated for the year 2020 (see `https://www.scimagojr.com/journalrank.php?category=2613`).

The *Journal of the American Statistical Association* stands out as having relatively robust requirements. The publication's guidelines 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 processed 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; 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 least strict 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)).

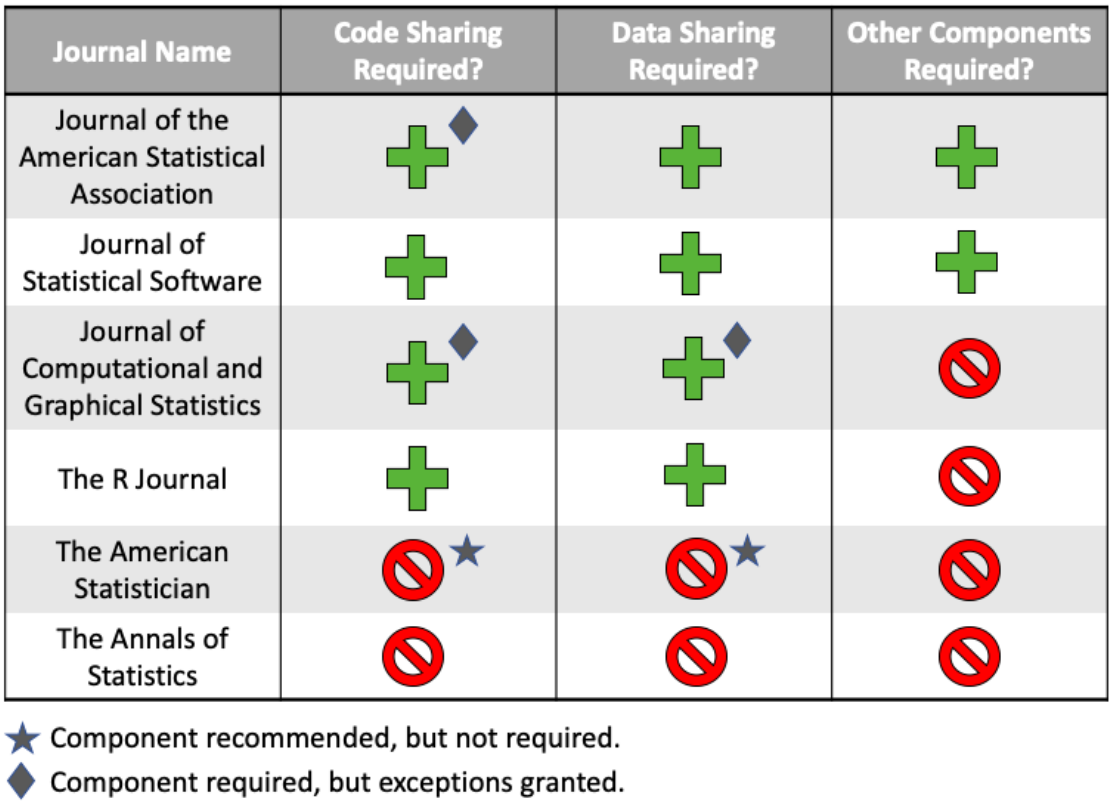


Figure 1.3: Reprodubility Policies of Top Statistical and Data Sciences Journals — JASA and JSS have the strongest reproducibility requirements, while The American Statistician and The Annals of Statistics have next to none.

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.

	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 1.4: The TOP Factor Rubric — The TOP Factor method uses an eight category rubric to measure reproducibility in the sciences.

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.

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)).

1.4.4 Assessing the Success of Academic Reproducibility Policies

We have seen that, although not necessarily the standard, some journals from 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 implemented 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 supplemental 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) compared the reproducibility of published work both before and after the journal *Cognition* instituted an open data policy, that required authors to 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 contrast 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.

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 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 importance of 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 a very time and labor intensive process, as we will see in the example of the *American Journal for Political Science* (AJPS), whose reproducibility policy was discussed previously 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 re-submission (the average number of re-submissions 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.

Reviewers themselves are also not inclined to support an increased focus on reproducibility. AJPS is an exception, but Journal review is often not a paid position. In many cases, reviewers are obligated to participate as part of their expected contribution to academia. They receive no compensation for their efforts and every additional hour of labor is an unwanted burden. Any changes that add complication to the review process are not desirable; although reproducibility may be important, reviewers are not likely to support a focus on it unless the process is streamlined.

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 challenges over legality of sharing data, large commitments of time or money, difficulties in 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 programs (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 to 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 seem natural to students. If learned independently, without effective tools, it can be challenging and even disheartening to try to understand and succeed at achieving reproducibility. 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 have 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 (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 statistical 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; and
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 1.5: Popular Reproducibility Packages in R — Drake, packrat (now renv) and workflowr are some of the more well-known reproducibility-focused R packages.

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 1.6: Popular Continuous Integration Tools — Many of these tools are integrated into other tools, like GitHub, making them easily accessible.

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 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. Their significant learning curve can also drive away 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 user-friendly and lack the ability to address features specific to certain programming languages.

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 developed.

Chapter 2

`fertile`: My Contribution To Addressing Reproducibility

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

`fertile` is an `R` package that attempts to combine the 7 key features of an effective reproducibility tool, discussed at the end of Chapter 1, into a singular piece of software that can be easily downloaded, installed, and operated by `R` users.

Figure 2.1: fertile's Package Logo

`fertile` attempts to address the gaps in existing reproducibility 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 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 6 major components of reproducibility. We will consider `fertile`'s treatment of each of these components in turn, exploring its behavior using the sample R project shown below, titled `project_miceps`:

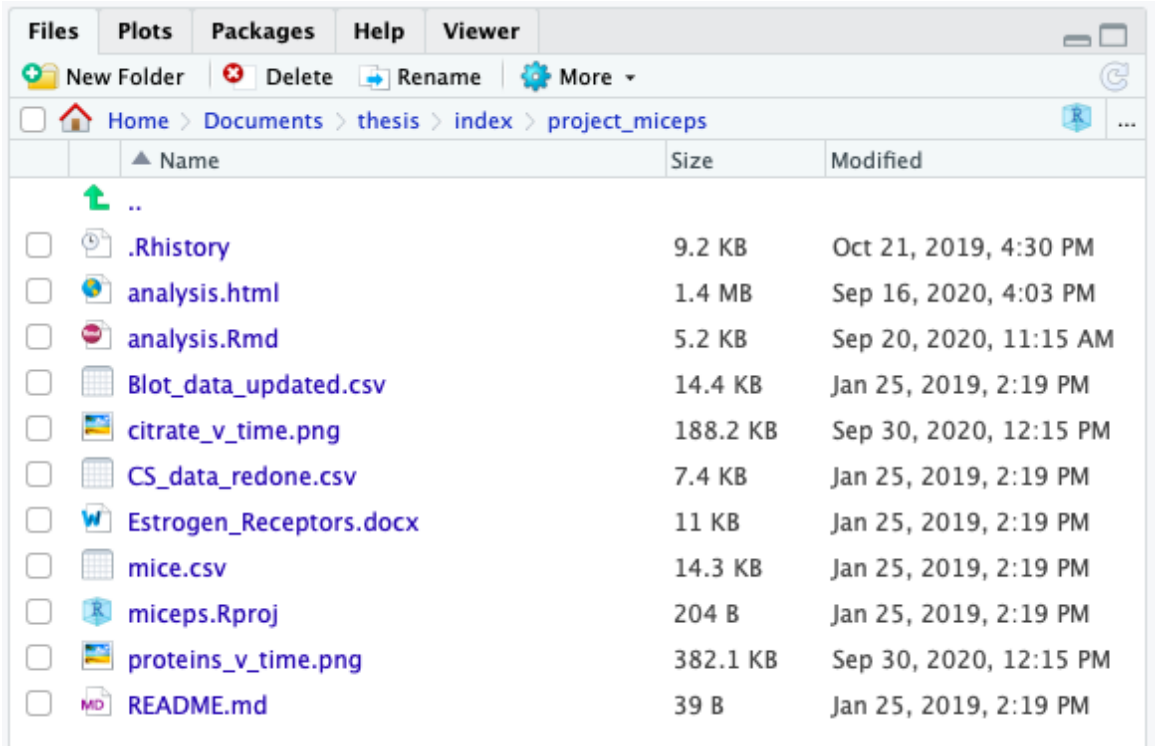


Figure 2.2: A Sample R Project

`project_miceps` contains data and analysis from a study concerning the differences in bicep muscle growth between male and female mice. The results from this study,

conducted in part by Smith College professor Stylianos Scordilis, were published in *Cellular and Molecular biology letters* in 2010 (Metskas, Kulp, & Scordilis (2010)).

The folder contains a variety of different file types: - 1 `.Rmd` file containing the analysis code. This involves data manipulation, visualization, and random number generation - 1 `.html` containing the knitted output of the analysis code - 3 `.csv` files containing the experimental data - 2 `.png` image files containing plot outputs from the analysis - 1 `README` file containing information about the project - A `.docx` file containing text about estrogen receptors in mice - `.Rproj` and `.Rhistory` files, containing data used by `R` to track work and maintain the project structure

In the following sections, we will consider the functionality of `fertile` as demonstrated primarily through the results of a reproducibility analysis conducted on `project_miceps`. All of the code for `fertile` that will be discussed in this chapter can be found at `https://github.com/baumer-lab/fertile`, while the local branch tracking my changes can be found at `https://github.com/ambertin/fertile`.

2.1.1 Component 1: Accessible Project Files

The inclusion of key files—such as analysis scripts, data, and documentation information—helps ensure reproducibility by providing the audience with access to everything relevant and necessary to the process of re-running a project.

```
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 way 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 auxiliary (image or text) files they have.

When this is run on `project_miceps`, we see the same files as described before– their names, file sizes, extensions, and a "mime," which summarizes the file type in a more general way than the extension.

```
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~  185.32K 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~  377.87K 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()`. Since, as we know, `project_miceps` contains a `README` file, this function returns a positive confirmation.

```
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—not dependent on any way in its location in a user's file system—it is important to test whether the project still executes properly if it is moved to another directory than its primary location in the user's computer.

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. This function works on both compressed (`.zip`) and non-compressed directories.

Before the files are sandboxed, we see them in their original directory.

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

```
project_miceps/Blot_data_updated.csv

project_miceps/CS_data_redone.csv

project_miceps/Estrogen_Receptors.docx
```

After they are sandboxed, we see that the directories for the files have changed, now beginning with a sequence starting with `/var/folders...`—the path used in temporary directories—indicating that the files have now been copied to a new location.

```
temp_dir <- sandbox('project_miceps')
fs::dir_ls(temp_dir) %>%
  head(3)
```

```
/var/folders/v6/f62qz88s0sd5n3yqw9d8sb300000gn/T/RtmplgM6bB/project_miceps/
Blot_data_updated.csv
```

```
/var/folders/v6/f62qz88s0sd5n3yqw9d8sb300000gn/T/RtmplgM6bB/project_miceps/
CS_data_redone.csv
```

```
/var/folders/v6/f62qz88s0sd5n3yqw9d8sb300000gn/T/RtmplgM6bB/project_miceps/
Estrogen_Receptors.docx
```

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.

Below, `proj_render()` executes without error, indicating that the project compiles successfully when removed from its original environment—a positive sign of reproducibility.

```
proj_render(temp_dir)
```

```
-- Rendering R scripts... --------------------------------- fertile 1.1.9003 --
```

Users can confirm the last time the project was known to have successfully rendered by checking the time of the last successful render, captured in the `render_log_report()` function. If the recorded time of last render was *before* the time at which `proj_render()` was run, this would indicate that the project did not compile successfully.

```
# Get last row of render log report (last rendered time)

render_log_report(temp_dir) %>%
  tail(1)
```

```
# A tibble: 1 x 4
  path          path_abs func        timestamp
  <chr>         <chr>    <chr>       <dttm>
1 LAST RENDERED <NA>     proj_render 2021-04-06 12:46:53
```

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. Sub-projects can cause challenges with version control, confusing software that is trying to track changes by making it unclear where to focus, and should be avoided. 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.

This definition of "tidyness" comes from the `R` package structure. `R` packages are required to follow a format with specific folders for specific files. Any miscellaneous files found at the top level of the package directory are flagged when building a package. Top-level clutter is not allowed, and files have to be moved into their respective folders.

This is not the only definition of tidyness. Some individuals may prefer a system where a variety of files *are* stored at the root directory of a project because this simplifies the method of writing paths to access these files—if everything is located in one place, you only need to provide the name of the file to open it. However, with `fertile`, we believe the `R` package structure to be advantageous. By recommending that users follow this package structure in the reproducibility recommendations, `fertile` makes it easy for users to convert their `R` projects into `R` packages with minimal effort. This can be useful for analysts whose work contains functions and datasets. The `R` package format makes function use very simple—functions are built in with package installation, rather than requiring users to execute `R` scripts to bring the functions into their environment. `R` packages also have a built in method of documentation, using `roxygen2`, that makes it easy to ensure that any functions are well-explained. The package format can also be advantageous for data sharing. If data is built into a package, it can be easily accessed using the `data()` command, rather than requiring users to use some version of `read.csv()` or any other data-reading function. By working to set up projects based around the package format, `fertile` sets users up to easily be able to achieve these benefits in their analyses if they are desired.

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 apparent 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 paths are 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.

When `has_only_used_files()` is run on `project_miceps`, two files are returned: the `.docx` file and one of the data files, `mice.csv`. This indicates that neither of those files were referenced in the code, nor were they created by it. Since no other files were returned, this indicates that all of the other files were either inputs or outputs to the data analysis.

```
# 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
```

```
# 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 that help with reshaping the entire project structure to a more reproducible format. For example, `proj_suggest_moves()` provides suggestions for how to reorganize the files into folders, with recommendations based on an R package structure.

When run on `project_miceps`, the results indicate that the `.csv` files should be moved to a `data-raw` file, the `.Rmd` should be moved to vignettes, and the `.docx`, `.html` and `.png` files should be moved to an `inst` folder, with further deliniation by file type.

```
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~  185.32K 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~  377.87K 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, Boettiger, & Mullen (2018), and believed by me, 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. It also provides the advantage of making it easy to convert the project into a package, if necessary.

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 essential 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. According to Prana, Treude, Thung, Atapattu, & Lo (2019), alongside a variety of best-practices posts from `R` coders, 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. As discussed in the previous section on Component 1, this returns a positive result since `project_miceps` contains a `README` file.

```
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 and no error messages result:

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

```
v Checking for clear build chain
```

Compare this with the following example: `project_transportation`. This project contains three code files—`cars.Rmd`, `planes.Rmd`, and `trucks.Rmd`—with no clear order. None of the files have a number attached to them, nor is there a makefile in the directory.

```
proj_analyze_files("project_transportation")
```

```
Warning: Please include a README file in /Users/audreybertin/Documents/thesis/
index/project_transportation
```

```
# A tibble: 3 x 5
  file                                    size ext   mime            path_rel
  <fs::path>                       <fs::bytes> <chr> <chr>           <fs::path>
1 project_transportation/cars.Rmd          211 Rmd   text/x-markdown cars.Rmd
2 project_transportation/planes.Rmd        295 Rmd   text/x-markdown planes.Rmd
3 project_transportation/trucks.Rmd        188 Rmd   text/x-markdown trucks.Rmd
```

When `has_clear_build_chain()` is run on `project_transportation`, it fails, returning a note stating that "it is not obvious in what order to run [the] R scripts."

```
has_clear_build_chain("project_transportation")
```

```
* Checking for clear build chain

   Problem: It is not obvious in what order to run your R scripts

   Solution: Use a formal build chain system or prefix your files with numbers

   See for help: ?drake::drake
```

```
# A tibble: 3 x 2
  culprit                          expr
  <fs::path>                       <chr>
1 project_transportation/cars.Rmd  ""
2 project_transportation/planes.Rmd ""
3 project_transportation/trucks.Rmd ""
```

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. As we see, the project loaded the `stargazer`, `skimr`, `purrr`, `ggplot2`, `tidyr`, `readr` and `dplyr` packages, indicating that they are potential dependencies. We can also see the R version and operating system version that the project was run on, as well as the fact that the rendering was completed on a Mac.

```
-- Rendering R scripts... --------------------------------- fertile 1.1.9003 --
```

```
The R project located at '/Users/audreybertin/Documents/thesis/index/
project_miceps' was last run in the following software environment:


R version 4.0.4 (2021-02-15)
Platform: x86_64-apple-darwin17.0 (64-bit)
Running under: macOS Big Sur 10.16

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_2.1.3      purrr_0.3.4
ggplot2_3.3.3    tidyr_1.1.3      readr_1.4.0      dplyr_1.0.
```

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.

When run on `project_miceps`, we find that `broom`, `dplyr`, `ggplot2`, `readr`, and other packages are all referenced in the code. As indicated by the installation script, these packages are hosted on CRAN and can be installed using `install.packages()`. We also see that there are no GitHub packages used in this project.

```
install_script <- proj_pkg_script("project_miceps")
```

```
# 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 (likely) hosted on GitHub...
```

As good practice, users interested in ensuring that their project is reproducible should include both of the following files alongside it.

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 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. This 10% recommendation is designed to be a low-bar recommendation, setting its minimum on the lower end of comment production, but within a reasonable distance from the mean. It is based on data from Arafat & Riehle (2009), who found the mean percentage of comments in open source code was around 18%, with a standard deviation of approximately 10%. This places a 10% comment minimum within 1 standard deviation below the mean.

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()`, indicating that the creator of that file should add more comments to clarify what is happening in the code. In addition, it recommends a link to a site containing tips for adding comments to 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 comments.

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

# 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")`, even though both paths point to the same location in the computer.

```
library(fertile)
```

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

```
[1] TRUE
```

When we try to access the `my_data.csv` file via an absolute path to the computer desktop, where the file is located, `fertile` recognizes that an absolute path has been provided and returns an error stating this. It also gives the option for the user to run `utils::read.csv('~/Desktop/my_data.csv')`, which will not throw an error, but will return the same result.

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

```
Error: Detected absolute paths. Absolute paths are not reproducible
and will likely only work on your computer. If you would like
to continue anyway, please execute the following command:
utils::read.csv('~/Desktop/my_data.csv')
```

The error message is different when referring to a relative path that is outside of the directory. In the example below, we reference the same `my_data.csv` file, but instead using relative paths indicating to go back three directories to find the `/Desktop` folder. This time, the `fertile` warning message mentions path that "lead outside the project directory." The suggested override command also changes to reflect the path that was provided.

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

```
Error: Detected paths that lead outside the project directory. Such paths
are not reproducible and will likely only work on your computer. If you
would like to continue anyway, please execute the following command:
readr::read_csv('../../../Desktop/my_data.csv')
```

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 more reproducible way.

```
getwd()
```

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

```
setwd("project_miceps")
```

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

The `here::here()` suggestion is primarily for users writing code scripts, rather than executing from the console, who may find it helpful to set their directory in code—for example, to ensure that an RMarkdown file executes the same when run chunk-by-chunk as when knit. The `here` package makes it easy to access files when executed as part of a script. The function works by finding the path to a file relative to the top level of your project directory. This path is always in a relative format associated with the location the script file is located it, making it impossible to write a non-reproducible absolute path that links to a user/computer-specific location.

`here::here()` is not necessary for console-based programming, nor is `setwd()`. As long as users use the R project format, which is required by `fertile`, and all files for the analysis are kept within the project directory, all paths can easily be written as relative to the project root without the need for `here::here()` or `setwd()`. This is one of many strengths of the R project format: it negates the need for reproducibility-breaking functions in interactive programming. Users who new to reproducibility

and working in the project format, however, may still need reminders not to use `setwd()`. It is for this reason that `fertile` provides an error when this function is called, regardless of whether it was done interactively (where there is no need for it), or in a script (where setting the directory could potentially seem advantageous).

For more information about the interactive warning system and some of its customization features, see Section 2.2.1.

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.

When this function is run on `project_miceps`, we see no paths at fault, indicating that all paths in the project are reproducible.

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

```
-- Rendering R scripts... ---------------------------------- fertile 1.1.9003 --

-- Generating reproducibility report... -------------------- fertile 1.1.9003 --

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()`.

For example, since the current working directory (`getwd()`) is always returned as an absolute path, running `check_path()` on it throws an error warning about this fact.

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

```
Error: Detected absolute paths. Absolute paths are not reproducible
and will likely only work on your computer.
```

If providing a path leading outside the project directory, up a level in the file framework, an error mentioning paths outside the project directory is returned.

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

```
Error: Detected paths that lead outside the project directory. Such
paths are not reproducible and will likely only work on your computer.
```

These functions, together, cover all of the reproducibility sub-components 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 one or more functions using random number generation were called. Finally, in order to select an appropriate response, `fertile` determines the type of randomness that has occurred.

Here, I make a distinction between two primary types of randomness in `R`: *controlled* (or semi-random) and *uncontrolled*. All randomness in `R` is pseudo-random and not truly random—the number generation system uses algorithms to generate sequences of random numbers which approximate a truly random outcome but are not in fact themselves completely random—but not all randomness is controlled. The distinction between controlled and uncontrolled randomness emerges from the use of `set.seed()`. When `set.seed()` is used alongside pseudo-random number generation, the number generator returns the same "random" sequence of numbers each time. In a sense this is "random," because it is still a pseudo-randomly generated sequence, but it is also non-random in that it is predictable and repeatable—and therefore, controlled. Every time the code is run, it will generate the same sequence of numbers. This is in contrast to the behavior when `set.seed()` is excluded from the code, what I term *uncontrolled* randomness. In this situation, functions employing random number generation will *not* return the same pseudo-random sequence each time they are run. This will therefore affect the outcome of results and the reproducibility of the work. Two people running the same code when randomness is *uncontrolled* will not receive identical outputs, while they would if the randomness were *controlled*. For this reason, `fertile` flags *uncontrolled* randomness as dangerous, but has no problem with *controlled* randomness.

Let us consider an example of how `has_no_randomness()` performs on `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.

```
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")
  ))
```

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

However, there is no seed set in the file. `project_miceps` is using *uncontrolled* randomness. As a result, `has_no_randomness()` returns as a failure for the project.

```
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)
```

If a seed had been set, however, this code would have passed. Consider this alternate example. The following sample project, `project_randomseed` contains only a `.Rproj` file and an `.Rmd` document containing the following code:

```
set.seed(1)
rnorm(2)
```

In this situation, randomness occurs in the calling of `rnorm()`, but it is *controlled* by the use of `set.seed`. This time, `has_no_randomness()` passes without error:

```
has_no_randomness('project_randomseed')
```

```
v Checking for no randomness
```

`fertile`'s randomness-centered feature helps analysts know that their use of randomness 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 a 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 Wickham's "tidy" code format, which promotes the following best practices (Wickham (2020)):

- 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',
# Test has been wrapped so it can all be seen at once

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.

If written in tidy style, this code would look as follows:

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

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.

After being converted to tidy style, the code looks as follows:

```
if (length(mice) > 1) {
  holder_var <- 1
}
```

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 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 1.1.9003 --

 [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

When `proj_analyze()` is run on `project_miceps`, we see the following information: - 10 packages (including `broom`, `dplyr`, `fs` and more) are referenced in the analysis code - 12 files are in the project directory, including some image files, data files, and a few files produced from some of the analyses that have been run by fertile (`install_proj_packages.R` and `software-versions.txt`). - There are 10 suggestions for moving files, with data files to `data-raw`, script files to `R`, and other files to `inst`

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

```
# 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
# A tibble: 12 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     185.66K image/png
 3 proteins_v_time.p~ png     380.89K 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 install_proj_pack~ R           260 text/plain
10 software-versions~ txt       5.34K text/plain
11 miceps.Rproj       Rproj       204 text/rstudio
12 analysis.Rmd       Rmd        5.21K text/x-markdown
# A tibble: 10 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.cs~
 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.d~
 4 analysis.Rmd       vignettes  file_move('project_miceps/analysis.Rmd', fs::d~
 5 analysis.html      inst/text  file_move('project_miceps/analysis.html', fs::~
 6 citrate_v_time.png inst/image file_move('project_miceps/citrate_v_time.png',~
 7 install_proj_pack~ R          file_move('project_miceps/install_proj_package~
 8 mice.csv           data-raw   file_move('project_miceps/mice.csv', fs::dir_c~
 9 proteins_v_time.p~ inst/image file_move('project_miceps/proteins_v_time.png'~
10 software-versions~ inst/text  file_move('project_miceps/software-versions.tx~
NULL
```

Similar to `proj_check()`, `proj_badges()` runs all sixteen reproducibility checks. 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()`, `_images()`, `_code()`, `_raw_data()`, `_data()`, `_scripts()`, `_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.

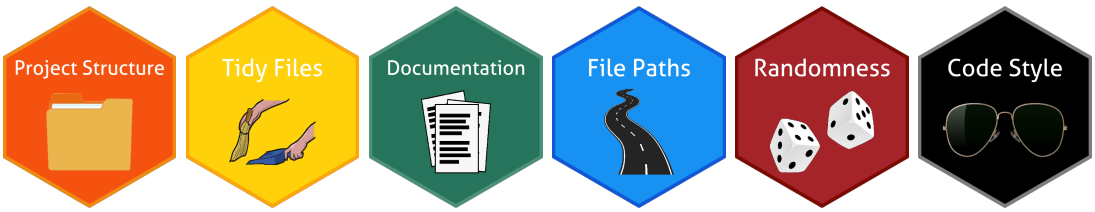


Figure 2.4: The 6 Reproducibilty Badges Available in 'fertile'

`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:

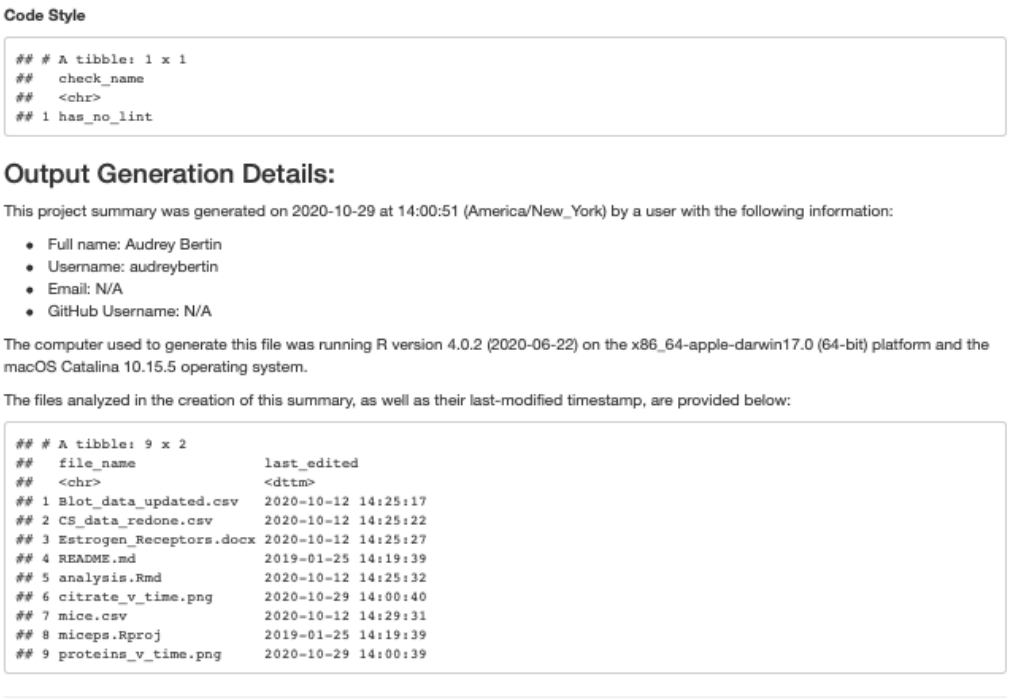
1. The name of the folder where the project is stored.
2. The reproducibility badges awarded to the project.
3. The reproducibility badges failed by the project.
4. If failures occurred, the reasons why. These are broken down by badge and list the checks associated with that badge that did not pass.
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. As we can see, `project_miceps` receives the "Project Structure" and "File Paths" badges but fails "Tidy Files," "Documentation," "Randomness," and "Code Style." This aligns with our previous results from each of the components of reproducibility. For example, `project_miceps` passed `proj_analyze_paths()` with flying colors, but failed `has_well_commented_code()` (documentation), `has_no_randomness()` (randomness), and `has_no_lint()` (code style).

We see these failures reflected in the badge report section titled "Reasons for Failure," which provides the list of functions responsible for why the project failed to receive that badge. For example, we can see that for the "Tidy Files" badge, `project_miceps` failed `has_tidy_images()`, `has_tidy_raw_data()`, and `has_only_used_files()`.

At the end of the report, in the section titled "Output Generation Details," we see information similar to what was produced with the previously described dependency management functions. We see the date the project was rendered by me, Audrey Bertin, on October 29th, 2020 at just after 2 pm Eastern time using R version 4.0.2 on a Mac computer running macOS Catalina. Additionally, we see the files contained in the project and their last modified dates.

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()` or `proj_check_badge()` functions.

`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. When run on `project_miceps`, we see that three of the tidyness checks fail, while three pass. The errors are associated with tidy scripts, image files, and data files.

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

```
-- Running reproducibility checks -------------------------- fertile 1.1.9003 --

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 1.1.9003 --

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. When we do this, we see that `project_miceps` has no issues with project roots and has a good root structure.

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

```
-- Running reproducibility checks -------------------------- fertile 1.1.9003 --

v Checking for nested .Rproj files within project

v Checking for single .Rproj file at root level

-- Summary of fertile checks ------------------------------- fertile 1.1.9003 --

v Reproducibility checks passed: 2
```

`proj_check_badge()` is similar. It also allows users to run checks in groups, but these are not called by similar names, but rather based on which of the six components of reproducibility they belong to. This is done via a `badge` argument. Users input the shorthand name of the badge they want to run the checks for—e.g. "randomness," "paths," "structure," etc.—and `fertile` then executes the corresponding functions.

For instance, if a user wants to run all of the checks associated with documentation, they would run the following code:

```
proj_check_badge("project_miceps", "documentation")
```

```
v Checking for clear build chain

v Checking for README file(s) at root level

* 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

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

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

-- Summary of fertile checks ------------------------------ fertile 1.1.9003 --

v Reproducibility checks passed: 2

* Reproducibility checks to work on: 1

* 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
```

As can be seen above, this outputs the results from the following `fertile` checks involving documentation:

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

A user who wanted to check `project_miceps` for tidyness instead would run the same function, but changing out the second argument from "documentation" to "tidy-files":

```
proj_check_badge("project_miceps", "tidy-files")
```

```
Joining, by = "path_abs"

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

* 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 1.1.9003 --

v Reproducibility checks passed: 4

* Reproducibility checks to work on: 3

* 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~

* 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
```

This runs all of the checks associated with ensuring that there is minimal file clutter: the six checks starting with `has_tidy_`, alongside `has_only_used_files()`.

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 sub-component 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 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 (Epstein et al. (2002)).

Users interested in looking back at their past choices 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.

When we execute the commands `library(purrr)`, `library(forcats)`, and `read_csv(fs::path('project_miceps', "mice.csv"))`, we see those actions appear in the log. Since `read_csv` takes a file path argument, `fertile` captures the path that was called and reports it under `path`.

```
log_report()
```

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

```
# A tibble: 1,329 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 1,319 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. Few 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 reproducibility, including those that read and write files, load libraries, and set random number generation seeds.

When users perform actions that may threaten reproducibility, the package's shimmed functions capture the user's commands, log their behavior, and check and report on file paths before proceeding to executing the desired function.

This allows `fertile` to warn users when they use a potentially non-reproducible file path and also helps keep track of past behavior via a log of previously entered commands.

The shim-writing process looks as follows:

1. Identify an `R` function that is likely to be used in a way that may break reproducibility. For example, a function that reads in files.

2. Create a function in `fertile` with the same name that takes the same arguments.

3. Write this new function so that it:

    a) captures the specified arguments,
    b) logs the name of the function the user was intending to call,
    c) performs file path checks on any path-related arguments, and
    d) calls the original function with the original arguments.

If the file path checks execute without error, the operation of a shim looks the same to the user as if they were calling the original function. However, if an error is thrown—i.e. a path is found to be non-reproducible—then execution will stop before step *d* is completed.

Most shims are relatively simple. Several, such as that for `library()` are more complex, but many follow the same basic format. Consider this example for `read.csv()`:

```
fertile::read.csv
```

```
function(file, ...) {
  if (interactive_log_on()) {
    log_push(file, "utils::read.csv")
    check_path_safe(file, ... = "utils::read.csv")
    utils::read.csv(file, ...)
  }
}
<bytecode: 0x7f8330adbe48>
<environment: namespace:fertile>
```

This functionality all occurs without the knowledge of the user. Consider the example of `read.csv()`. `read.csv()` is a popular function from the `utils` package for reading in data files. Users with both `utils` and `fertile` loaded will experience the following: The user will call `read.csv()` as normal, thinking that they are calling `utils::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 `utils::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 `utils::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.

A full list of the shimmed functions provided by `fertile` is listed below. Note that—in order to reduce the software dependencies of `fertile` the shims from the less common `readxl`, `rjson`, `foreign`, and `sas7bdat` packages are only activated if the user has those packages installed and loaded on their computer.

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'

Shim Customization Functionality

Not all R programmers use the same packages, nor do they use the same functions at the same level of regularity. Additionally, some users might not always want to have their shimmed functions enabled—for example, they may purposefully be writing code with absolute paths, and would want to be able to do so without warnings and interruptions.

As a result of this, we believed it important to include features in fertile that enable users to customize which functions for which their version of fertile has active shims as well as when those shims are enabled or disabled.

Several functions in fertile have been written to implement this behavior:

1. The add_shim() function allows users to write a new shim to fertile. When this function is called, fertile writes the code for the shimmed function using the standard structure explained previously. It then adds that code to a file located alongside fertile's documentation on the user's computer so that it is permanently saved, and then executes it in order to activate the new shim and immediately bring it into the user's R environment. With add_shim(), users can only add shims individually, as in the example below, where a shim for the function write.ftable from the stats package is added.

```
# Code to add a shim for stats::write.ftable
add_shim(func = "write.ftable", package = "stats")
```

2. If users want to not limit themselves to a handful of specific shims and rather just automatically create all of the possible shims, the find_all_shimmable_functions(), find_pkg_shimmable_functions() and add_all_possible_shims() functions can assist.

find_all_shimmable_functions() searches the namespaces of all of the loaded libraries in search() and returns the names and associated packages of all functions found to potentially be shimmable (i.e. those that appear to accept file path-related arguments). This list provides a reference for those who want to know which functions they could potentially write shims for.

find_pkg_shimmable_functions() does the same thing as find_all_shimmable_functions(), but on a smaller scale. Rather than looking at all loaded packages, it just looks at the namespace of a specific package, passed as an argument to the user.

add_all_possible_shims() will look at the list of functions provided by find_all_shimmable_functions() and perform the actions of add_shim() on all of them individually. This will generate and then add all of those functions to the shim file included alongside the fertile documentation and then load them into the environment.

```
# Return list of all potential shims in ALL loaded packages
find_all_shimmable_functions()
```

```
# Return list of all potential shims from a SPECIFIC package (e.g dplyr)
library(dplyr)
```

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

```
find_pkg_shimmable_functions(package = "dplyr")
```

```
[1] "src_sqlite"
```

```
# Write/load all of the shims from find_all_shimmable_functions()
add_all_possible_shims()
```

3. By default, all of the shims in `fertile`—both those included in the package as well as those written by the user—are activated when `fertile` is loaded (using `library()`) and deactivated, i.e. removed from the environment, when it is unloaded.

There is no method currently in the package to deactivate the shims included in `fertile` without unloading it. However, the shims written by the user are completely controllable in this respect. `load_shims()` will find the file of user generated shims and execute it to bring those shims into the environment. `unload_shims()` will search the environment for functions that are linked to `fertile`, via unique attributes written into the functions, and remove them.

4. Finally, users can edit the file containing the shims they have written using the function `edit_shims()`. This will open an interactive window which provides the user with access to the file so that they can delete or reorganize shims as desired.

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:

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

```
[1] "project_miceps/.fertile_log.csv"
[2] "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 time stamp of the last time the code was run by `fertile`.

In the render log report for project miceps, we see calls to the packages that have previously been noted as dependencies (e.g. `dplyr`, `readr`, `tidyr`, etc.), references to file paths in the code (e.g. `Blot_data_updated.csv`), and a timestamp of the last render.

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

```
# A tibble: 21 x 4
   path                    path_abs                  func    timestamp
   <chr>                   <chr>                     <chr>   <dttm>
 1 Seed @ Start            <NA>                      612     2021-04-06 12:47:15
 2 package:dplyr           <NA>                      base::~ 2021-04-06 12:47:15
 3 package:readr           <NA>                      base::~ 2021-04-06 12:47:15
 4 package:tidyr           <NA>                      base::~ 2021-04-06 12:47:15
 5 package:ggplot2         <NA>                      base::~ 2021-04-06 12:47:16
 6 package:purrr           <NA>                      base::~ 2021-04-06 12:47:16
 7 Seed Before             <NA>                      612     2021-04-06 12:47:16
 8 /Users/audreybertin/Do~ /Users/audreybertin/Docu~ readr:~ 2021-04-06 12:47:16
 9 Seed After              <NA>                      3       2021-04-06 12:47:16
10 Seed Before             <NA>                      3       2021-04-06 12:47:16
# ... with 11 more rows
```

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—a variable whose value is stored in a user's local `R` environment, and whose value can be accessed or updated via commands—called`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 capture 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—meaning that they are both relative and point within the project directory—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
```

```
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: 0x7f832130bba0>
<environment: namespace:fertile>
attr(,"req_compilation")
[1] TRUE
```

2.2.4 The Dots (...)

`fertile` utilizes the dots (...) in shims, the file path messaging system, and the `proj_check_some()` function.

The dots are a mechanism built into `R` to allow for variability in the arguments passed to a function. They allow the user to pass in a variety of arguments to a function—in any number—rather than forcing them to pass in a set number. Additionally, the dots can be easily captured and passed on to another function.

These dots help `fertile` achieve its desired functionality in several ways.

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.

```
formals(readr::read_csv)
```

```
$file


$col_names
[1] TRUE

$col_types
NULL

$locale
default_locale()

$na
c("", "NA")

$quoted_na
[1] TRUE

$quote
[1] "\""

$comment
[1] ""

$trim_ws
[1] TRUE

$skip
[1] 0

$n_max
[1] Inf
```

```
$guess_max
min(1000, n_max)

$progress
show_progress()

$skip_empty_rows
[1] TRUE
```

`read_csv()` is not alone. Many functions like it have a large number of arguments. It would be a very cumbersome process to have to re-write all of them arguments by hand in each shim. Additionally, it would make the shim-writing process not very reproducible: if every shim took a lengthy list of arguments with all different names and types, which all had to be specified in the definition of each shim, it would be complicated to scale up the library of available shims! There would be no easy method to automate this process, so many would have to be written by hand.

When the dots are used, none of these problems arise. In a `fertile` shim, we only have to define the parameters that are required for the function being shimmed to operate—plus, of course, anything related to file paths—and then just say that everything else will go through the dots.

When a user executes a shim, anything passed through the path arguments gets evaluated for reproducibility and everything else is saved through the dots. Those dots then get handed over to the original function. This function knows what to do with them, because the user was running the `fertile` shim as if it were the original function, and therefore passed arguments that the original function could interpret.

In the `read_csv()` example, we see that the `fertile` shim takes only `file` and `...` as arguments. Users can still pass any of the 14 `readr::read_csv()`arguments besides `file` to the shim, and these are all saved as the dots.

After `fertile` checks the file path passed to `file` (using `check_path_safe`), it reads in the data using the original `readr` function, executing `readr::read_csv(file, ...)` which passes the file path independently, and then everything else the user specified—if anything—through the dots.

```
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: 0x7f83370418c8>
<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.

Unfortunately, while the dots are advantageous in shim-writing, they do come with a downside. In the original, un-shimmed versions of functions, the arguments are all defined by name. This can prove useful when users are coding interactively; while calling a function, help boxes often pop up summarizing all of the possible arguments and their default values. In the `fertile` shims, the only included arguments are path arguments, other required arguments, and the dots. As a result, these are the only ones that appear in the help box when calling a shim, and the full argument list is lost—users have to rely on their own knowledge or package documentation to execute their function, and cannot rely solely on the help box.

Additionally, 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 data frame, then passed through `dplyr::select(...)`, where the dots contain the information about the user's `tidyselect` call. `tidyselect` is written in such a way that it knows how to process different strings of commands passed through the dots. When `fertile::proj_check_some()` user passes a string of commands understood by `tidyselect`, `tidyselect` interprets them and then runs them on a dataframe containing the names of the checks, allowing for custom check selection using the `tidyselect` framework.

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()`:

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). This customization is meant to help with dependency tracking. `proj_badges()` outputs change to display custom information about the computer type and operating system version of the machine on which the code was run, which—in addition to detailed information about the project itself—helps insure complete information about all potential factors which could be influencing software behavior.

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. Tables containing information why the failed badges were not achieved (the names of the badge-specific tests that failed), and
3. A table ontaining 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`.

2.3 A Note About This Behavior

It is important to note that the behavior of `fertile`, described in the previous section, *How `fertile` Works*, could potentially be considered by some as a type of malware:

- `fertile` creates hidden files on the user's computer. Although the package does not overwrite existing files and only creates files for the purpose of allowing itself to function at full capacity, this hidden file behavior might still be considered intrusive, as these files are written *outside* of the directory where `fertile` is downloaded.

- `fertile` could also be described as "tricking" the user. The use of masking/shimming, in a way, is a method of convincing the user to think one thing is happening—for instance, that they are calling `readr::read_csv`—while really,

something else is actually happening—they are calling `fertile::read_csv`. This is a type of deception which could be considered undesirable.

- This shimming behavior comes with the added concern that users do not know what is happening under the hood while they are executing commands. Theoretically, the package *could* be tracking their personal user data, such as an IP address, and sending it to a server somewhere, but the user would not know this in the moment.

These concerns are a major part of why `fertile` is not hosted on CRAN. Its potentially-malware-like behavior is likely to be considered too intrusive for acceptance onto the platform.

However, these malware-like features are what makes `fertile` so functional and provides it with the reproducibility features that are not present in any other software at this time.

It is, of course, though, important to recognize and alleviate these concerns as much as possible. This is one reason—among many others—that `fertile` is hosted on GitHub. All of the code for the package is publicly available, making its behavior completely transparent. By looking through the code, users can see where hidden files are getting created, when they are getting overwritten, and what happens behind the scenes on each of the shimmed functions. Nothing is left to mystery.

Additionally, users have the opportunity to decide for themselves whether they want to use the software. `fertile` is not built into R by default. Users must take action—by researching packages, installing `fertile`, and loading it into their `R` environment—to activate the package. In big part due to the transparency around the code, at any step in this process users have the agency to review what they are getting into and decide for themselves whether they want to use the software.

2.4 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 shallow 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 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. `fertile`'s methods are not the most rebust solution to dependency tracking—packages like `packrat` focus specifically on this purpose and offer more

functionality—but they do help move users in the right direction.

Another R-specific feature that `fertile` addresses is randomness. Randomness is 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 Section 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.

Additionally, as discussed in Section 2.2.1, the interactive warning system is also highly customizable, providing users with functions to write their own shimmed functions that check for file paths and enable or disable those as desired.

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

Chapter 3

Incorporating `fertile` Into the Greater Data Science Community

Addressing reproducibility on a widespread scale is a challenging problem. In academic publishing, software, and data science education some progress has been made but many approaches 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.

OR

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 quickly 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 through only a handful of functions, all of which are simple.

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. 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 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 several 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 authors 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 reproducible 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 a 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.

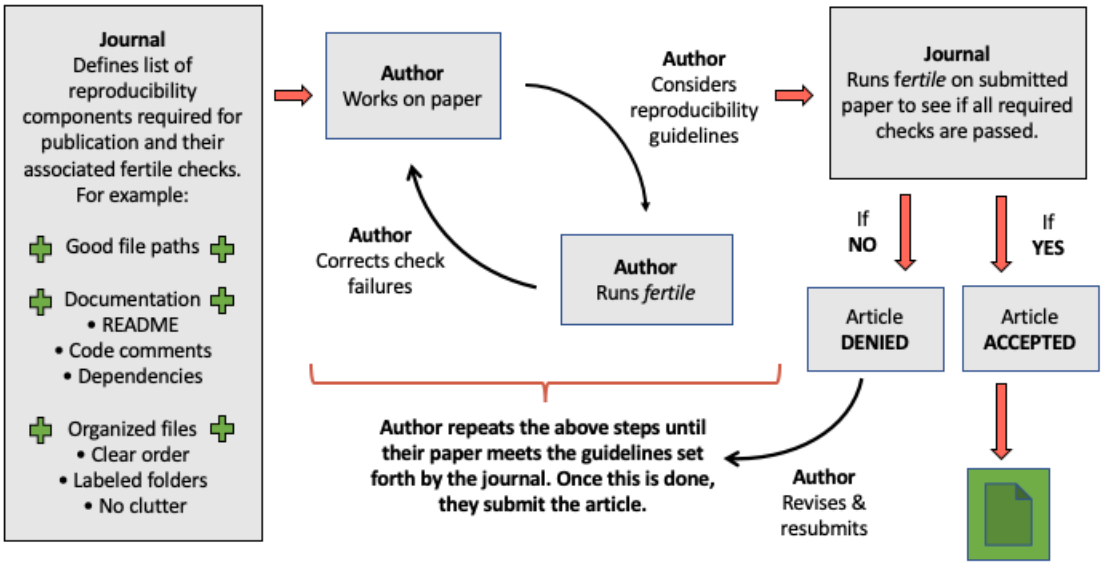


Figure 3.1: Potential 'fertile' Journal Review Process

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 cover sheet—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 cover sheet. This cover sheet 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.

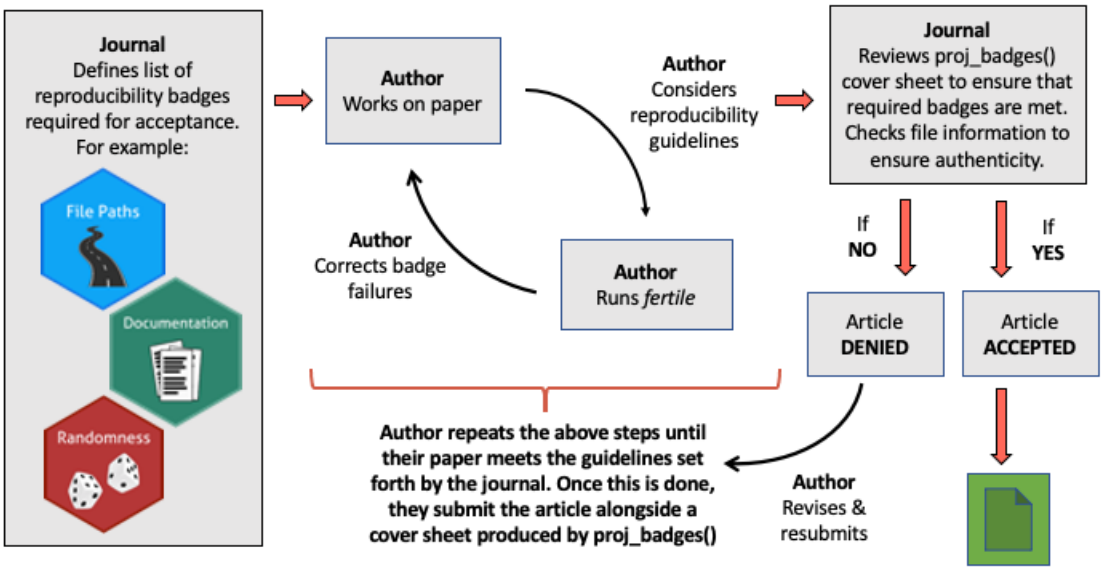


Figure 3.2: Another Potential 'fertile' Journal Review Process

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 authors 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 save significant time and money for 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.

Undergraduate students do not have the same level of access to reproducibility education. Some schools prioritize reproducibility education–for instance, introductory courses at Smith College and Duke University actively integrate reproducible workflows into their course material through the use of RMarkdown (Baumer, Cetinkaya-Rundel, Bray, Loi, & Horton (2014))–but not all do, and most (if not all) not at as in-depth of a level as a replication-based masters course might provide.

`fertile` could help change this, allowing for many more colleges and universities to integrate reproducibility into their courses as a primary topic of focus. The barriers to entry for using and benefiting 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.

5 Replication Paper Assignment

For graduate students enrolled in Gov2001, the main class assignment is to write a research 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 undergraduate 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 depending 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 members 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 (outside 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 uploading 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 Professor Gary King's Gov 2001 Graduate Course

`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 teachable 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.

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

- Teaching reproducibility early on could help give 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 could afford them an opportunity to fail and learn without fear of judgment.
- Integrating concepts early through repetitive use and recall helps ingrain them in the minds of students, ensuring that reproducibility begins to come naturally to them (Karpicke & Roediger (2008)).

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

3.1.3 In Other Areas

`fertile` can also provide benefits in other domains. Though not an exhaustive 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!

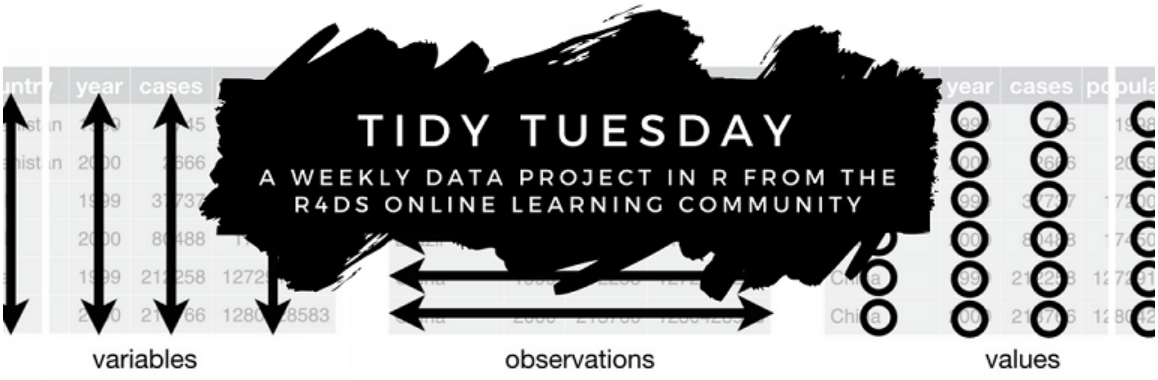


Figure 3.4: The Logo For Popular Tidy Tuesday Event

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

3.2 Testing fertile in the Real World: Experimental Design and Analysis

3.2.1 Background

Many of the suggested uses for `fertile` in the previous section are hypothetical, included for the purpose of demonstrating the potential that the package could have. Without experimentation or testing, though, it is difficult to know just how successful `fertile` would be when employed for any given purpose.

In order to gain a better understanding of the practical effects of using `fertile` in a real life scenario, Professor Baumer and I decided to conduct an experiment (approved by Smith College IRB, Protocol #19-032). We were interested in testing whether the integration of `fertile` into an introductory data science course might have an effect on students' learning of reproducibility concepts.

After designing our experimental structure, working out technical mechanics, and creating the necessary materials, we chose to use the following two course sections for our study:

1. Introduction to Data Science (SDS 192), Fall 2020 Semester
2. Introduction to Data Science (SDS 192), January-Term 2021

There were several motives for selecting these specific courses:

- They were both being taught by Prof. Baumer, which made it easy to ensure that everything went smoothly by removing the need to interact with a 3rd party.
- They were two different sessions of the same course, with essentially identical material, ensuring that we could combine results from each section of the experiment.
- They involved introductory data science students, one of the groups that were of most interest as we wanted to ensure that `fertile` could provide benefit to users of all skill levels.
- Though certainly not large sample sizes by any measure, they were still somewhat larger than many other classes in the Statistical and Data Sciences department, providing an opportunity for more data collection.

3.2.2 Experimental Design

For the experimental structure, we chose to complete a randomized controlled trial (RCT) since RCT designs can be used to conclude causality and are designed in such a way as to limit bias in the results.

Our goal was to determine whether students who had `fertile` installed and loaded on their R Studio applications would learn more about reproducibility throughout the course than students who did not, or whether there was instead no difference or even a negative effect.

In order to measure this potential effect, we opted to use reproducibility "tests," forms containing multiple choice and select-all-that-apply questions about different aspects of reproducibility that are focused on in `fertile`.

We used two different testing structures for our two experimental sections–partially due to challenges which led to our first experiment session starting on a slight delay, but also so that we could compare which structure worked more effectively. While the primary goal of the experiment was to determine the effect of `fertile`, a secondary interest was in the experimental method itself–a new method that we developed, which has implications beyond purely this case.

- Structure #1 (Used in Fall 2020): Students received a reproducibility test at the end of the semester to test their knowledge after the course. Final scores were compared between students who had `fertile` installed and loaded and those who did not.

- Structure #2 (Used in January Term 2021): Students received a reproducibility test both at the beginning of the course *and* at the end of the course in order to test their *change* in knowledge. *Differences in scores* were compared to see if any given group learned more, less, or the same.

Although the testing structures were different across the two different sections, the reproducibility tests were identical.

To reduce bias, we opted to use blinding, in which participants were unaware of whether they had received the experimental or control condition–in this case, whether they had `fertile` installed and loaded while working on course assignments, or whether they didn't.

To achieve the desired blinding, we decided to use some features of R packages which allowed us to give all students in the course the same software, but have it behave differently when loaded, depending on which experimental group the student was a part of. This technique is described in the step by step experimental structure, explained below:

1. Students were given a form at the start of the semester asking whether they consented to participate in a study on data science education. Additionally, information was collected about the previous number of courses that student had taken in data/computer science fields, which we believed might impact their knowledge of reproducibility. In order to successfully consent, students had to provide a unique computer identifier, collected through the command `whoami::username()`. This identifier was collected in order to ensure that students could be individually identified and therefore assigned effectively to treatment or control groups.

2. To maintain student privacy, the unique identifiers were then transformed into hexadecimal strings via the `md5()` hashing function. To the students' computers, and us—the researchers—these hashed strings contained the same information as the identifiable usernames, but to those without the original identifier, they appear only as a meaningless string of letters and numbers.

3. These hexadecimal strings, which correspond to unique students, were then randomly assigned into equally sized groups, one experimental group that receives the features of `fertile` and one group that receives a control.

To ensure an adequate control, with the control group students operating day-to-day in an almost identical way to the experimental group, we created a software package for the course which every student in the class would use, regardless of group. The package, titled `sds192`, contained several templates and data sets that were necessary

for use in the course, so all students in the class–regardless of their experimental status–needed to install it.

The `sds192` package was designed to look the same to both the experimental and control group students, but to function slightly differently for each. The control group would only have access to the templates and data provided with the `sds192` package, and the experimental group would have all of those things *plus*, they would have `fertile` silently loaded in the background. This meant that `fertile`'s proactive features were enabled, and therefore experimental group users would receive warning messages when they used absolute or non-portable paths or attempted to change their working directory. Control group users, who did not have `fertile` loaded for them, would not receive these messages.

This control/experimental group differentiation works by leveraging an `.onAttach()` function in the `sds192` package to scan the `R` environment of each student *each time the package is loaded*, collect their unique identifier–again, via `whoami::username()`– and run it through the same hashing algorithm as before.

`sds192` came pre-coded with the hashed identifiers of all of the students, grouped by whether they had been assigned to the experimental or the control group, to facilitate the operation of `.onAttach()`. Each time a student's hashed identifier was collected (every time they loaded the `sds192` package for class), it was then compared with the provided groups to determine whether the behavior of the package would match the control or experimental group.

The structure of this function can be seen below:

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

.onAttach <- function(libname, pkgname) {

  # The experimental group gets `fertile` loaded silently
  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",
    # ... and so on
```

```
  )
  control_group <- c(
    "9aa36583f54766205850428e8f1a4c89",
    "f03020938b31818063c79d2422755183",
    "7ec57b1f2bca9ac1e702fb68427b781b",
    "e5e30623e9d09d29ded851b7fb40cb51",
    "592572bb9fce168f37117fd0d6e0e5ee",
    # ... and so on

  )


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

}
```

4. [January Term 2021 Only] Students then took a reproducibility test prior to fully starting the course, which tested their knowledge before completing the class.

5. The students then completed their coursework as normal, regularly using the `sds192` package for projects and homework assignments.

6. [Both Fall 2020 and January Term 2021] The last step was for students to take a reproducibility test at the end of the course, to measure their knowledge of the subject after having completed the coursework and used the `sds192` package.

3.2.3 Results

In order to understand the results, it is important to understand the structure of the experimental reproducibility "test" and the way in which each student's answers were scored.

The test consisted of two primary sections:

1. A set of six questions on RStudio projects, two of which were select-all-that-apply, and four of which were multiple choice. These questions tested students knowledge of RStudio projects, their understanding of which file paths they could include in code submitted in project format, and their awareness of the dangers of changing working directories while coding.

2. A set of three questions about file paths, all of which were select-all-that apply. These questions tested students' knowledge of the difference between absolute and relative paths and their understanding of which paths are reproducible.

For reference, the full list of questions is provided in Appendix B.

For each test submission, every question was scored individually and then scores for each section (projects and paths) were calculated, as well as an overall total score.

The following method was used to compute scores:

- For each question, students would start at a score of zero. Based on their answers, points would then be added, subtracted, or stay constant.
- On select-all-that apply questions, the following scoring rules were used. This meant that students who answered some things correctly, but other things incorrectly, could still receive negative or neutral scores, based on the overall analysis of how correct they were:
  - Each correct box that was checked –> +1 Point
  - Each incorrect box that was checked –> -1 Point
  - Each box not checked but should have been –> No change (0)
- On multiple choice questions, the scoring rules were slightly different. This was due to the inclusion of an option for "I'm not sure," which was included as a way to discourage random guessing and ensure that students were answering based on their actual knowledge.
  - If the correct answer was selected –> +1 Point
  - If the incorrect answer was selected –> -1 Point
  - If the student expressed uncertainty ("I'm not sure") –> No change (0)

This method meant that individual questions could have negative, positive, or neutral scores, and total scores could vary quite dramatically, from large negative scores (<-10) for students who regularly answered incorrectly to large positive scores for those who often answered correctly (>10).

Fall 2020 Results

The data from Fall 2020, after being scored and cleaned up, looked as follows:

```
glimpse(fall2020)
```

```
Rows: 18
Columns: 15
$ User_Hash        <chr> "\"67b8da3952e07ba3f2c5c715c4042220\"", "\"f03020938b~
$ Group            <chr> "Control", "Control", "Fertile", "Fertile", "Fertile"~
$ Previous_Classes <dbl> 0, 0, 2, 1, 2, 1, 6, 0, 1, 1, 1, 2, 1, 1, 5, 1, 3, 1
$ Q1_Points        <dbl> 3, 1, 1, 3, 3, 2, 3, 3, 3, 1, 1, 3, 1, 1, 2, 2, 3, 1
$ Q2_Points        <dbl> 0, -1, 0, -1, -1, 1, -1, 1, -1, 0, -1, -1, 0, -1, 1, ~
$ Q3_Points        <dbl> 0, 1, 0, -1, -1, 0, -1, 1, -1, 0, -1, -1, 0, 1, 1, 0,~
$ Q4_Points        <dbl> 1, 1, 1, 1, 1, 0, 1, -1, 1, 0, 1, -1, 0, 1, 1, -1, 1,~
$ Q5_Points        <dbl> -1, 1, 0, 1, 1, 1, 0, 1, 1, 0, -1, 1, 0, -1, 1, -1, 1~
$ Q6_Points        <dbl> 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, -1, 1, 0
$ Q7_Points        <dbl> 2, 2, 3, 0, -1, 3, 3, 3, -1, 2, 2, 2, 2, 2, 3, 1, 2, 2
```

```
$ Q8_Points        <dbl> -1, 2, -2, 2, -2, 2, 2, 2, -2, 2, 1, 1, -1, 1, 2, -1,~
$ Q9_Points        <dbl> 1, 1, 0, 2, 2, 2, 2, 2, 1, -1, 1, -2, 0, 1, 1, 1, 2, 2
$ Projects_Total   <dbl> 3, 4, 3, 4, 4, 5, 2, 5, 4, 1, -1, 1, 1, 2, 7, -2, 4, 2
$ Paths_Total      <dbl> 2, 5, 1, 4, -1, 7, 7, 7, -2, 3, 4, 1, 1, 4, 6, 1, 5, 5
$ Overall_Total    <dbl> 5, 9, 4, 8, 3, 12, 9, 12, 2, 4, 3, 2, 2, 6, 13, -1, 9~
```

There were 18 students who completed the reproducibility test, split evenly into two groups of 9, and 15 variables per student. Since the Fall 2020 section of the experiment contained only one reproducibility test, each student only has one associated observation in the data.

1. `User_Hash`: Hashed identifiers, representing unique students.
2. `Group`: A categorical variable summarizing whether the student was in the experimental (`fertile`) or control group.
3. `Previous_Classes`: The number of previous data science / computer science courses the student had taken at the start of the semester. 4-12. `Qx_Points`: The number of points scored on each individual question.
4. `Projects_Total`: The total number of points scored on the six questions associated with R Projects.
5. `Paths_Total`: The total number of points scored on the three questions associated with file paths.
6. `Overall_Total`: The total combined score of all nine questions.

The de-identified data, including all 15 variables of interest, are available with open access at `https://github.com/ambertin/thesis`.

The score averages for both groups of students were as follows:

```
# A tibble: 2 x 4
  Group   Projects_Avg Paths_Avg Overall_Avg
  <chr>          <dbl>     <dbl>       <dbl>
1 Control         3.33      4.22        7.56
2 Fertile         2.11      2.44        4.56
```

These numbers, combined with the visual below seem to indicate, at least in this small sample, that the `fertile` group actually performed worse overall than the `control` group.

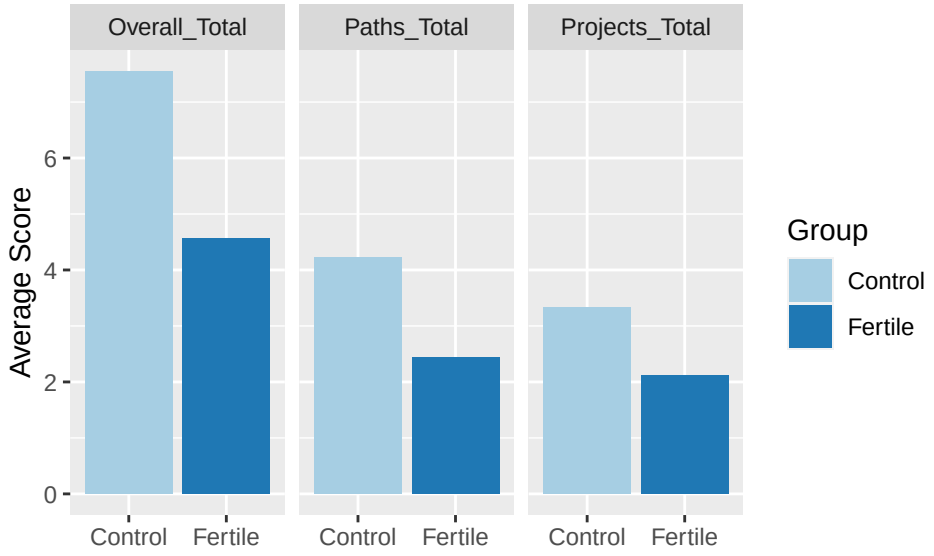


Figure 3.5: Bar Chart: End of Course Reproducibility Test Averages by Group, Fall 2020

Looking at the overall total score (`Overall_Total`), this was true even when controlling for the number of relevant courses taken previously:

```
`geom_smooth()` using formula 'y ~ x'
```

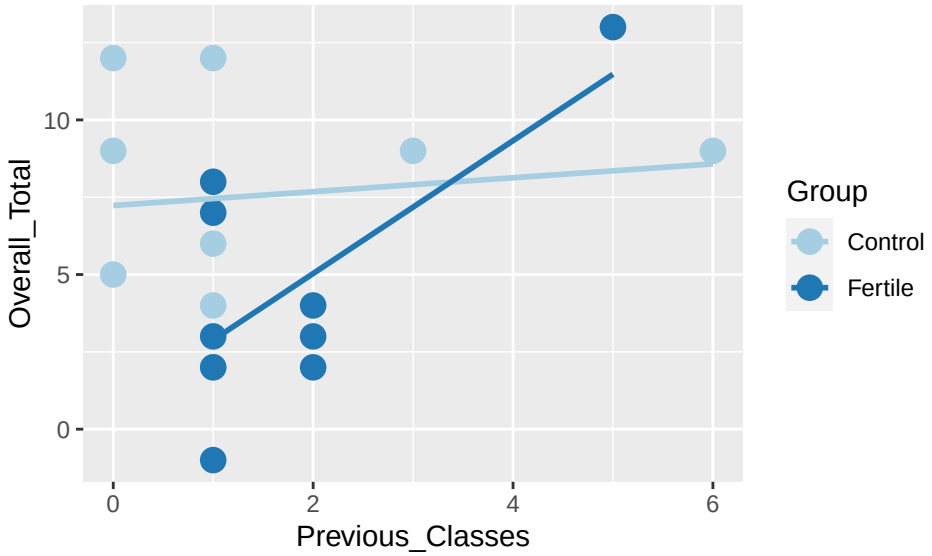


Figure 3.6: Scatterplot: Previous Coursework vs Total Reproducibility Test Score, Fall 2020

A t-test and analysis of variance (ANOVA) conducted on the simple regression model $Overall_Total \sim Group$ both indicate that, although there is a potential negative trend between the use of `fertile` and a student's score on the reproducibility test, there is not enough evidence to conclude that this relationship was *not* just the product of

chance. The p-values of 0.117 are not below the threshold of 0.05 used to determine statistical significance.

```
Call:
lm(formula = Overall_Total ~ Group, data = fall2020)

Residuals:
   Min     1Q Median     3Q    Max
-5.556 -2.556 -1.056  2.194  8.444

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)     7.556      1.281   5.896 2.25e-05 ***
GroupFertile   -3.000      1.812  -1.655    0.117
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.844 on 16 degrees of freedom
Multiple R-squared:  0.1462,    Adjusted R-squared:  0.09288
F-statistic: 2.741 on 1 and 16 DF,  p-value: 0.1173

Analysis of Variance Table

Response: Overall_Total
          Df Sum Sq Mean Sq F value Pr(>F)
Group      1  40.50  40.500  2.7406 0.1173
Residuals 16 236.44  14.778
```

When fitting a second model, controlling for the number of previous classes, we see a similar result. Although the p-value is smaller ($p = 0.08$), holding the number of previous classes constant, there is still not enough evidence to suggest that the use of `fertile` impacts reproducibility test scores.

```
fall_model_mr <- lm(Overall_Total ~ Group + Previous_Classes, data = fall2020)
summary(fall_model_mr)
```

```
Call:
lm(formula = Overall_Total ~ Group + Previous_Classes, data = fall2020)

Residuals:
   Min     1Q Median     3Q    Max
-5.191 -2.197 -1.055  2.969  5.803

Coefficients:
                 Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept)        6.3714     1.4803   4.304 0.000627 ***
GroupFertile      -3.2733     1.7613  -1.858 0.082835 .
Previous_Classes   0.8198     0.5615   1.460 0.164901
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.715 on 15 degrees of freedom
Multiple R-squared:  0.2525,    Adjusted R-squared:  0.1528
F-statistic: 2.533 on 2 and 15 DF,  p-value: 0.1128
```

January Term 2021 Results

We see a similar story with the data from January Term 2021, though the structure of the data set is different.

The January Term data looks as follows:

```
glimpse(jterm2021)
```

```
Rows: 76
Columns: 17
$ User_Hash        <chr> "\"96eb0673c18b9fa79740686995b1d970\"", "\"980a97cbb1~
$ Student_ID       <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16~
$ Group            <chr> "Control", "Control", "Fertile", "Fertile", "Fertile"~
$ Previous_Classes <dbl> 1, 1, 2, 1, 1, 4, 3, 0, 0, 1, 1, 4, 3, 4, 2, 4, 4, 0,~
$ Test             <fct> Pre, Pre, Pre, Pre, Pre, Pre, Pre, Pre, Pre, Pre, Pre~
$ Q1_Points        <dbl> 2, 2, 1, 2, 1, 3, 2, 2, 1, 2, 1, 2, 2, 1, 2, 1, 1, 2,~
$ Q2_Points        <dbl> -1, 0, -1, -1, 1, 1, 1, 0, 0, -1, -1, 0, 1, -1, -1, 0~
$ Q3_Points        <dbl> -1, 0, -1, 0, 0, 1, -1, 0, 0, -1, 0, 0, 1, -1, 0, 0, ~
$ Q4_Points        <dbl> 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, -1, 1, 1, 1, 1, 0, 1, 0~
$ Q5_Points        <dbl> 1, 0, -1, 1, 0, 1, -1, 0, 0, -1, 1, 0, 1, 1, 0, 0, 1,~
$ Q6_Points        <dbl> 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1~
$ Q7_Points        <dbl> 2, -2, 0, 0, 3, 2, 3, 1, 2, 3, 1, 2, 3, 3, 2, 2, -1, ~
$ Q8_Points        <dbl> 1, -3, -1, -2, 2, 1, 2, 0, 1, 2, 1, 1, 2, 2, 1, 1, -2~
$ Q9_Points        <dbl> 1, 2, 1, 1, 1, 1, -2, 0, -2, 2, -1, -2, 2, 2, -1, 1, ~
$ Projects_Total   <dbl> 3, 3, 0, 4, 3, 8, 2, 3, 3, 1, 1, 4, 7, 2, 3, 2, 6, 1,~
$ Paths_Total      <dbl> 4, -3, 0, -1, 6, 4, 3, 1, 1, 7, 1, 1, 7, 7, 2, 4, -5,~
$ Overall_Total    <dbl> 7, 0, 0, 3, 9, 12, 5, 4, 4, 8, 2, 5, 14, 9, 5, 6, 1, ~
```

It has many of the same features as the Fall 2020 data, though there are a few differences.

1. This data set contains data on 38 students, rather than 18.
2. Each student has two rows of data that correspond to them, rather than just one.
3. There are two new variables, `Student_ID` and `Test`. `Student_ID` is an index, used to keep track of which rows belong to the same student. This was created as an alternative to `User_Hash` as we found that users of Smith College-provided

laptop computers would occasionally have non-unique hashed IDs. `Test` has either the value "Pre" or "Post," identifying which test the row corresponds to–the test at the beginning of the course, or the one at the end, respectively.

Just as with the Fall 2020 data, these de-identified data are available publicly at `https://github.com/ambertin/thesis`.

When calculating the difference in reproducibility test scores between the end ("Post") and beginning ("Pre") of the course, the story looks a little bit less negative than before, but once again there does not appear to be much strong evidence in favor of `fertile`.

For the questions about file paths, there appears to be no difference in the average score at the beginning and end of the course for either group. For the questions about R Projects, the `fertile` group had slightly higher average score differences but not by that much (1.92). For the overall total number of points, the score differences were very similar (2.15 for the `control` group and 2.22 for `fertile`).

```
# A tibble: 6 x 3
# Groups:   Group [2]
  Group   Question_Group Avg_Score_Difference
  <chr>   <chr>                         <dbl>
1 Control Paths                          0
2 Control Projects                       1.2
3 Control Total                          2.15
4 Fertile Paths                          0
5 Fertile Projects                       1.89
6 Fertile Total                          2.22
```

When considering the distribution of test scores, in addition to just the pure difference, we see a pattern where those students in the `fertile` group appeared to score worse than the `control` group, on average. Yet, when considered all together, the `fertile` group's scores improved slightly more over the time of the course than the `control` group's.

This evidence appears to strengthen the previous finding–using the Fall 2020 data– that the potential negative relationship between reproducibility test score and use of `fertile` is likely a product of random chance. In both experimental sections, Fall and January Term, students in the `fertile` group performed slightly worse than the `control` group on the reproducibility tests taken at the end of the course. However, the additional evidence from the January Term "Pre-Test" appears to indicate that the reason for that lower score might be just that they started from a lower point of knowledge at the beginning, rather than that `fertile` directly affected their scores.

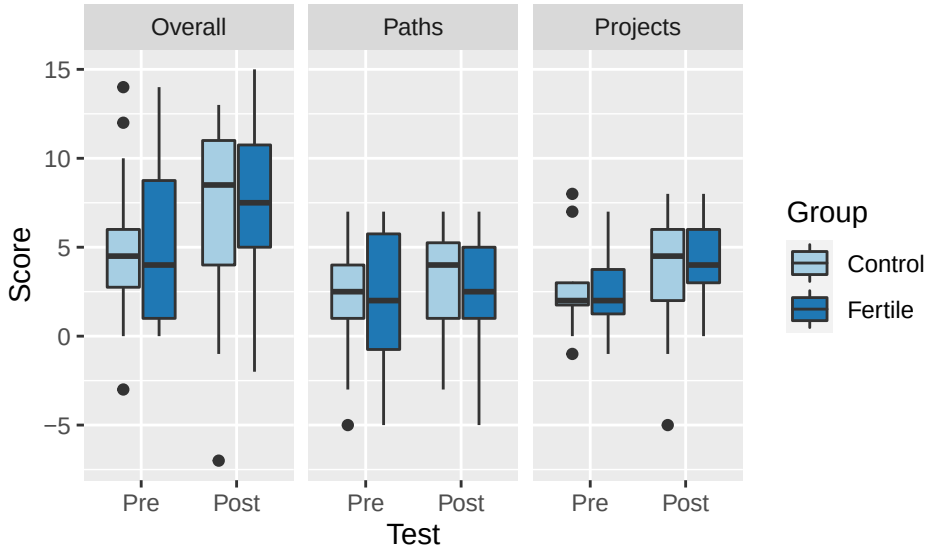


Figure 3.7: Boxplot: Pre- and Post-Test Score Distributions by Group, January Term 2021

It is also interesting to examine the relationship between the number of previous courses taken by students and their performance on the reproducibility tests.

Below, we see a graphical representation of the number of previous courses taken by each student and the difference between their overall total score on their Pre- and Post-Tests.

Although it is challenging to make any strong claims with so few data points, one interesting pattern emerges. For the `control`, group, previous experience did not appear to be an indicator of stronger performance or overall learning throughout the course. In some cases, overall performance of students with several previous classes was worse on the Post-Tests than the Pre-Tests. However, with the `fertile` group, there does appear to be a slight positive relationship between the number of previous classes and the test scores. On average, the `fertile` group always performed better on their Post-Tests than on their Pre-Tests.

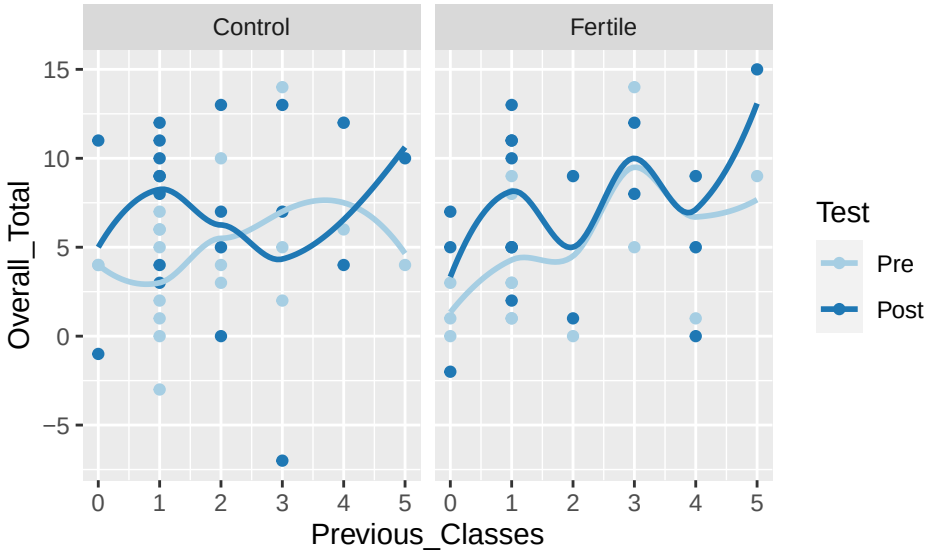


Figure 3.8: Scatterplot: Previous Coursework Experience vs Reproducibility Score by Group, January Term 2021

After constructing a simple regression model, predicting overall `Score_Difference` based on `Group` (`fertile` or `control`), and running a t-test and analysis of variance, we once again find that there is not enough evidence to argue that `fertile` provides any benefit in terms of performance on the reproducibility tests. This can be seen with the high p-value of over 0.96, indicating little evidence of a relationship between score differential and group.

```
Call:
lm(formula = Score_Difference ~ Group, data = jterm_model_data)

Residuals:
    Min      1Q  Median      3Q     Max
-14.150  -2.918   0.850   3.546   8.850

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   2.15000    1.10658   1.943   0.0599 .
GroupFertile  0.07222    1.60783   0.045   0.9644
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.949 on 36 degrees of freedom
Multiple R-squared:  5.604e-05, Adjusted R-squared:  -0.02772
F-statistic: 0.002018 on 1 and 36 DF,  p-value: 0.9644

Analysis of Variance Table
```

```
Response: Score_Difference
          Df Sum Sq Mean Sq F value Pr(>F)
Group      1   0.05  0.0494   0.002 0.9644
Residuals 36 881.66 24.4906
```

Adding in a second variable, the number of previous related courses taken by the student, we see the same story. Neither the student's experimental group, nor their number of previous classes have a statistically significant relationship with score difference, holding the other variable constant.

```
Call:
lm(formula = Score_Difference ~ Group + Previous_Classes, data = jterm_model_data)

Residuals:
    Min      1Q  Median      3Q     Max
-13.395  -2.130   1.043   2.996   8.232

Coefficients:
                 Estimate Std. Error t value Pr(>|t|)
(Intercept)       3.45431    1.52672   2.263    0.030 *
GroupFertile      0.06459    1.59648   0.040    0.968
Previous_Classes -0.68648    0.55789  -1.230    0.227
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.914 on 35 degrees of freedom
Multiple R-squared:  0.04152,   Adjusted R-squared:  -0.01325
F-statistic: 0.7581 on 2 and 35 DF,  p-value: 0.4761
```

3.2.4 Limitations

Although the findings of the `fertile` experiment are interesting, they come with many caveats and should be treated with caution.

For one, the sample sizes of both experimental sections were very small. The first only contained data on 18 students, while the second on only 38. While sample sizes like these may provide some evidence of the existence of experimental relationships, conclusive statements should not be made without additional data. Even though the SDS 192 courses are some of the largest in the Statistical and Data Sciences department at Smith College, they still did not result in an optimal level of data. Not all students volunteered to participate in the study, several who did failed to complete all of the components necessary for inclusion of their data and—even if everyone had participated—the maximum sample size would have only been limited to the combined size of both course sections: around 100 students.

Second, the fact that the environment for the experiment was an active classroom came with several challenges. Although we wanted students in the experimental group

to independently learn from `fertile` the classroom setting meant that students who had questions about some of the errors that appeared as a result of having `fertile` loaded would sometimes bring them to Professor Baumer. As the course instructor, he would then be impelled to help them, explaining concepts and clarifying the meaning of any errors, at the expense of any potential effect on the experimental results.

Additionally, students in a classroom have the unfortunate characteristic—in experimental terms—of not being fully independent from one another. Students regularly communicated with each other, asking and answering questions, on a course `Slack` channel and also worked in groups on many of the class projects. This meant that they could spread knowledge; experiences from the `fertile` group could be disseminated to the `control` group and vice versa. Any sort of cross-contamination could be a potential threat to the validity of the experimental results, as independence is a key assumption necessary for linear regression.

Although independence is at risk, many of the other assumptions for linear regression appear to be mostly met.

In the residuals versus fitted values and normal quantile-quantile plots for the Fall 2020 and January Term 2021 data, displayed below in that order, there do not appear to be any trends indicating an egregious lack of adherence to regression conditions.

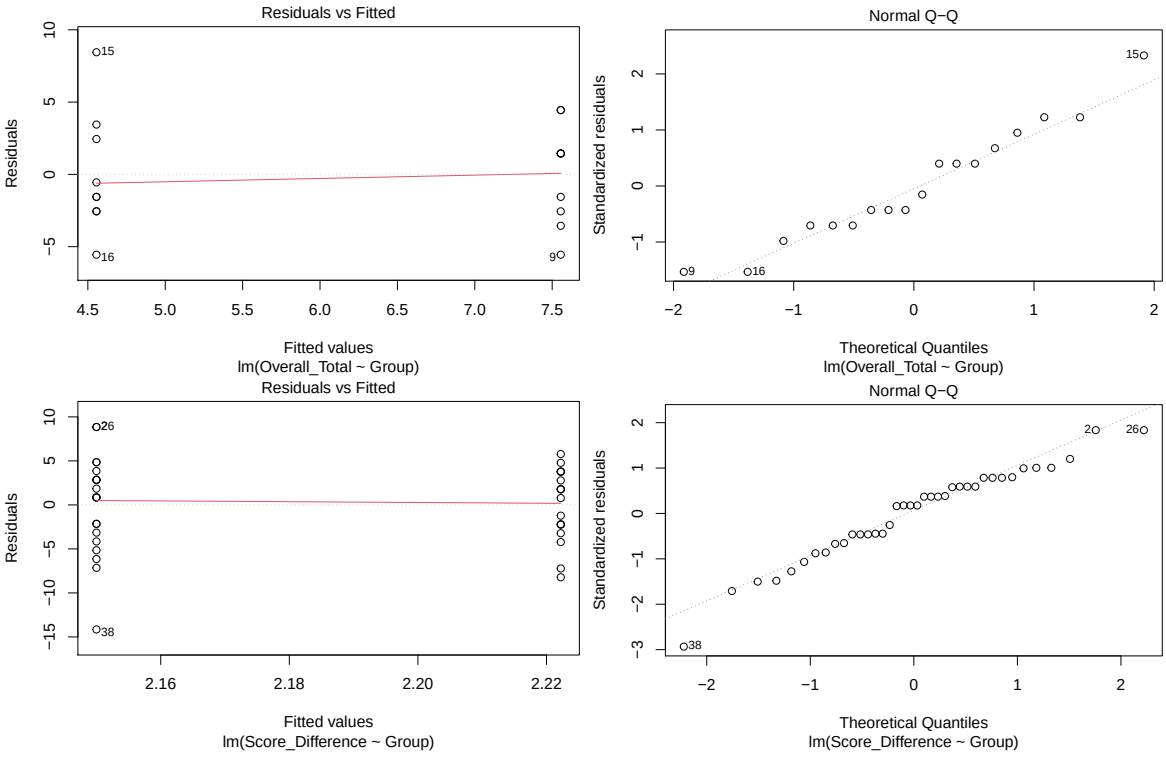


Figure 3.9: RvF and QQ Plots for Fall 2020 (Top) January Term 2021 (Bottom) Simple Linear Regression Models

Then, there is the additional limitation that, although this experimental structure tests the pure educational power of `fertile`, it does not accurately represent the ways in which `fertile` would be used in practice.

Those who use R packages for their own benefit do so with specific intention and knowledge that they are doing so. In practice, unless attempting to do some sort of software testing, no one will use a package that they do not know is loaded.

In order to create an effective experimental environment, however, concessions must be made. If `fertile` were tested in its natural use state, where students were aware that they were using it, it would introduce the possibility of significant bias in the results. Students in the `fertile` group, for example, might change their behavior, knowing that their software is designed to improve reproducibility, and pursue extra research into the topic, affecting their test performance. We do not know, from this data, whether `fertile` has significant benefits in its standard practice. All we can conclude from this study is that in the specific conditions that `fertile` was used in this course, there was no evidence that the package's interactive reproducibility messaging provides learning benefits when used without students' knowledge.

There is also the fact that the results of this experiment cannot be generalized beyond the Smith College Introduction to Data Science course. In no way is Smith's SDS 192 course representative of the wider undergraduate introductory data science community. Smith classes have a very different student makeup than courses at most other institutions with the distinct feature of being non-male-dominated, so they present a learning environment that is not necessarily comparable. Additionally, the SDS 192 course has its own unique curriculum, not identical to other courses at Smith or at other institutions. Due to the unique course material and teaching methods of SDS 192, students in the class could potentially be more or less primed to process messages about reproducibility compared with students in other courses or at other institutions. In order to gain a better understanding of `fertile`'s performance in the environments of other colleges and universities, additional studies would be necessary.

Finally, there is the issue of experimental monitoring. After assigning students to experimental and control groups in the `sds192 .onAttach()` function, there is no easy way to monitor that the students in the experimental group had `fertile` working properly. This would require looking at each student's computer one by one, having them execute potentially reproducibility-breaking commands, and observing the answers, which would be both time consuming and potentially threatening to experimental blindness. Even if `fertile` were working for an experimental group student at the beginning of the semester, that also does not guarantee that they had it installed for the rest of the semester. That student could have potentially gotten a new computer or changed their unique login name, affecting whether they were correctly recognized as members of the experimental group. There also could have potentially been other bugs on student computers that prevented `fertile` from loading. Due to these reasons, it is not possible to guarantee that all students in the experimental group had `fertile` loaded and functioning for the entire semester.

3.2.5 Implications

Although the conclusions from this experiment are not particularly demonstrative, the implications of the study extend far beyond just this software.

Implications for the Future of `fertile`

Through this experiment, we did not discover any significant relationship between students' blind use of `fertile`'s interactive warning system and their knowledge or learning of reproducibility across the time span of the Introduction to Data Science (SDS 192) course.

Additionally, this experiment does not test `fertile` in its natural environment—use by an individual who has specifically chosen to use it and studied the documentation in order to understand how it works—or consider the ways in which `fertile` fills current gaps in software that exist in the `R` community.

As the project continues to spread and be shared, we hope to find a dedicated community of users who find benefit in its software tools and see an increased shift toward reproducibility in the `R` community.

Implications for the Experimental Design in Other Applications

To the best of our knowledge, the experimental design we developed is the first of its kind—a structure designed to test the difference between two versions of `R` software without the experimental participants being aware.

Typically, `R` package testing is primarily anecdotal: a developer will release a software update, request testing and relevant feedback from users, and make edits based on those responses. This method can be effective, but it does not have the same scientific backing behind it as experimental testing does.

If this experimental design gained traction in the `R` community, it could pave the way for scientific A/B testing of `R` packages.

A/B testing is an experimental methodology focused on user experience. In an A/B test, designers compare two different versions of a product, randomly assigning participants to groups to determine which version they will receive. Using statistical hypothesis testing, the researchers attempt to identify which software version was most successful.

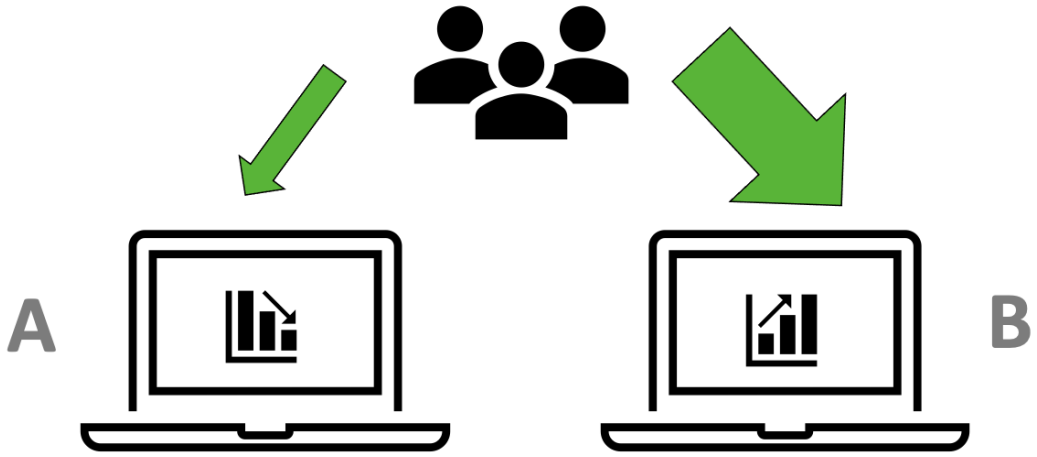


Figure 3.10: Scientific A/B Testing

One of the most popular applications of this form of testing is in marketing. Product sellers, in an attempt to drive traffic to their website and maximize profits, will test several different methods of advertising or web design to see which one results in the most clicks and/or purchase.

Though popular in the marketing industry, A/B testing has the potential to have a positive impact in any software-related domain, including `R` package development.

This will be particularly useful for those `R` developers looking to build a deeper scientific understanding of how to code more effectively, and may pave the way for more research-based improvements to software written in the `R` language.

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, we identify several components of 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 controlled.
6. Code is readable and consistently styled.

Reproducibility is vital 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. Many researchers across all scientific fields have been unable to reproduce each other's results, and some 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 are sub-optimal, facing challenges

with inaccessibility, steep learning curves, limited functionality, and/or lack of coding language-specific features.

`fertile` attempts to fill this gap by being:

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

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

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

4) Language specific, possessing features that address some of the reproducibility challenges associated with `R`.

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

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

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

Integration of `fertile` into the journal review process—through requirements for passing `fertile` checks or the inclusion of a `proj_badges()` report with a manuscript submission—would greatly reduce the barriers to reproducibility review, speeding up the review process and 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, introducing students to reproducibility much earlier in their data science careers than would likely happen otherwise and increasing the chance that they prioritize reproducibility in their future work.

`fertile` could also assist the general world of data science, through inclusion of popular events like Tidy Tuesday or integration into workplace environments, among other possibilities.

To test `fertile's` effectiveness in the classroom, we designed a randomized experiment on an introductory data science course to compare reproducibility learning between students who received `fertile` on their computer and those who didn't. Although no indication of a relationship between `fertile` use and knowledge improvement was found in this study—with the caveat that the study format, with blinding, did not fit the intended use of `fertile` in the real world—the experiment design itself has the potential to open up new avenues of code and package testing in the `R` community.

`R` programmers working in a variety of domains could use the experimental structure to employ A/B testing of their software, a type of scientifically-backed testing that has

previously not been used—at least to a notable degree—to test `R` code. This method of testing could pave the way for developers to scientifically measure which coding practices are most effective, efficient, and user-friendly.

Although it is difficult to predict the future of `fertile`, my hope is for this project is for it to help bring the `R` community, even if just a few users, toward improved reproducibility and to improve knowledge around the issue within the community. Anything beyond that—journal integration of `fertile` features, deployment of `fertile`'s A/B testing design, integration into the workflow by RStudio employees, or the like—would just be the cherry on top.

A Note on Limitations

Although this work focuses primarily on the advantages of `fertile`, it is important to recognize that `fertile` is not a be-all, end-all solution to reproducibility. `fertile` has its own set of limitations, including the following:

- Its work with dependency-management is far less advanced than some other software. Other packages like `packrat`, for example, go even further, allowing the user to build files containing exact copies of their dependency environment, and add/remove packages as desired.

- `fertile` has only been under development for a handful of years, much of which was part time by only one or two people as a side project. There are certainly features that would be helpful to be included in the package that have not been, simply because there were only so many things that could be reasonably achieved in the time the package was being written.

- `fertile` has around 30 software dependencies. This makes it highly susceptible to functionality-breaking changes if some of the functions `fertile` is dependent on are updated in a way that affects their operation. Unfortunately, this is one of the many trade-offs with `fertile`: in order to have a library of functions which address all different aspects of reproducibility, a variety of tools are needed, necessitating the use of many imports. This has been a challenge at several points in `fertile`'s history, requiring the rewriting of functions to address new bugs due to dependency updates. Though not as significant an issue during more recent months, it is likely that more problematic dependency updates will be released in the future.

- As described in Chapter 2, `fertile` could also be considered a sort of malware, executing operations without the user's knowledge and creating hidden files on their computer. This could be undesirable to some users.

For the last two reasons, `fertile` will likely never be available on CRAN, the official `R` package hosting site. Its malware-like behavior and high number of dependencies make the package potentially unstable and intrusive, features which are undesirable for packages. This makes `fertile` slightly less accessible to users who are unfamiliar with the process of installing packages from GitHub or that do not like doing so.

This thesis itself, too, has some limitations. It does not discuss every potential alternative to `fertile`, many of which have advantages over `fertile` that should be recognized. It also does not go into detail about every choice made during package development, which could leave out information of interest. Additionally, much of the work in Chapter 3, on potential applications is purely theoretical—there is no evidence of how `fertile` might actually work in a journal environment or in a classroom under non-experimental conditions. Claims about likely effectiveness are purely based on functionality, and would require much more testing and wide-spread deployment to be backed-up.

Even with all of these limitations, I believe that `fertile` has a lot of power to do good in the field of reproducibility. If it manages to help even just a few individuals by improving the reproducibility of their work, that is a success.

Appendix A

A Summary of Relevant Links

Several links relevant to `fertile` were shared throughout the text. To make them easier to find and review, they are all provided together below.

1. GitHub repository for `fertile`: `https://github.com/baumer-lab/fertile`
2. My GitHub fork of `fertile`, where you can track my changes and code contributions more easily: `https://github.com/ambertin/fertile`
3. GitHub repository for this thesis, which contains the sample project `project_miceps`, all of the non-R-generated figures that were included, as well as the de-identified data from the experiment: `https://github.com/ambertin/thesis`
4. DOI for the paper about `fertile` which was published in `Stat`: `http://dx.doi.org/10.1002/sta4.332`

Appendix B

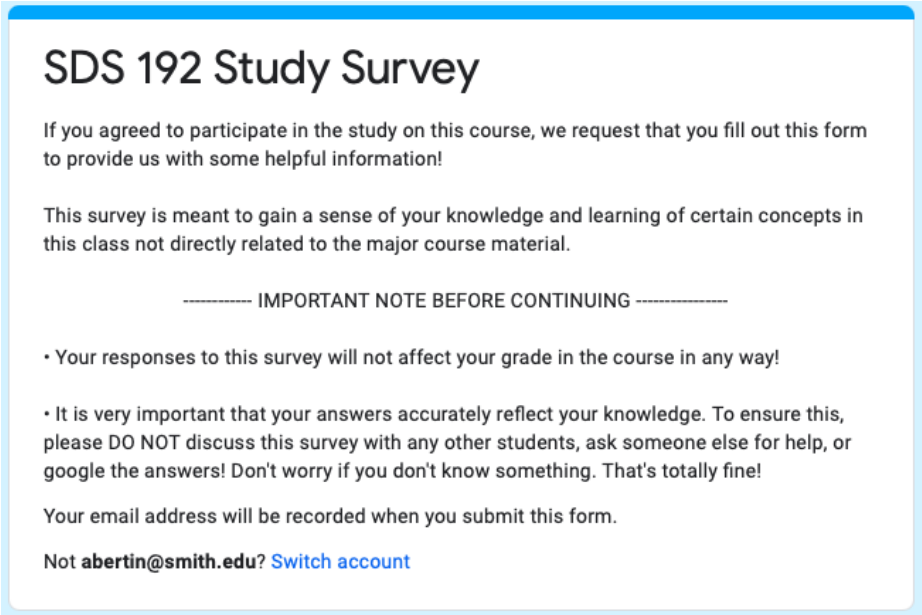
Full Overview of the Experimental Reproducibility Test

Chapter 3 discusses an experiment that Professor Baumer and myself conducted at Smith College to test `fertile` in a classroom environment.

In order to measure the package's effectiveness, we constructed a test that could capture information about students' knowledge of reproducibility. The full list of questions on that test, alongside with a step-by-step example of the scoring process, is provided here:

B.1 Full Reproducibility Test

Test Introduction



SDS 192 Study Survey

If you agreed to participate in the study on this course, we request that you fill out this form to provide us with some helpful information!

This survey is meant to gain a sense of your knowledge and learning of certain concepts in this class not directly related to the major course material.

---------- IMPORTANT NOTE BEFORE CONTINUING ----------

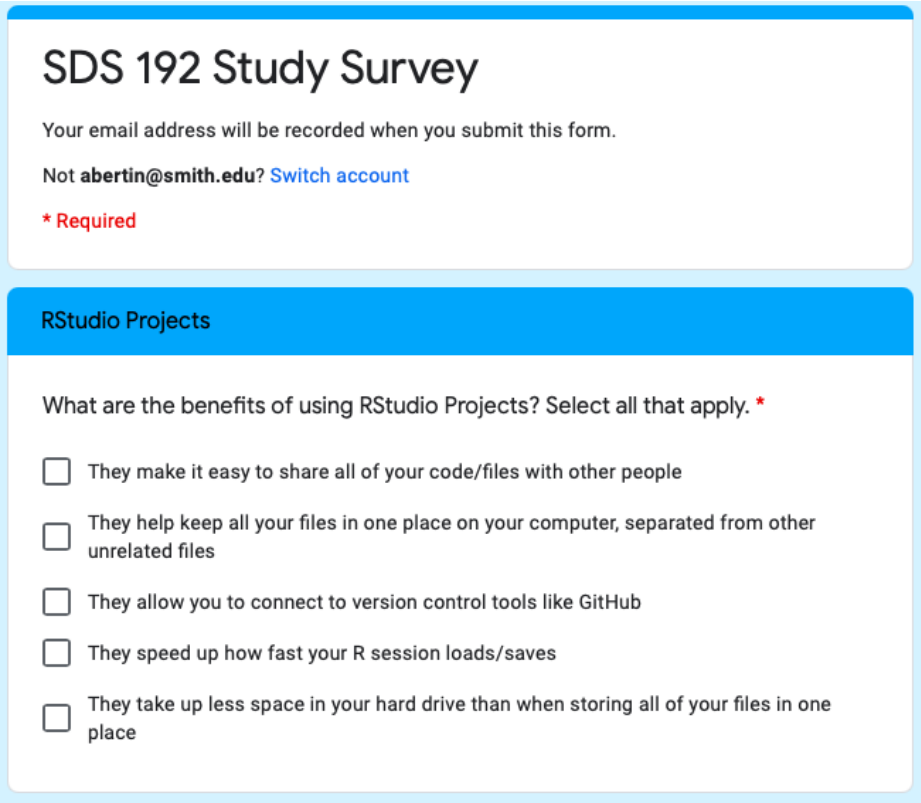
• Your responses to this survey will not affect your grade in the course in any way!

• It is very important that your answers accurately reflect your knowledge. To ensure this, please DO NOT discuss this survey with any other students, ask someone else for help, or google the answers! Don't worry if you don't know something. That's totally fine!

Your email address will be recorded when you submit this form.

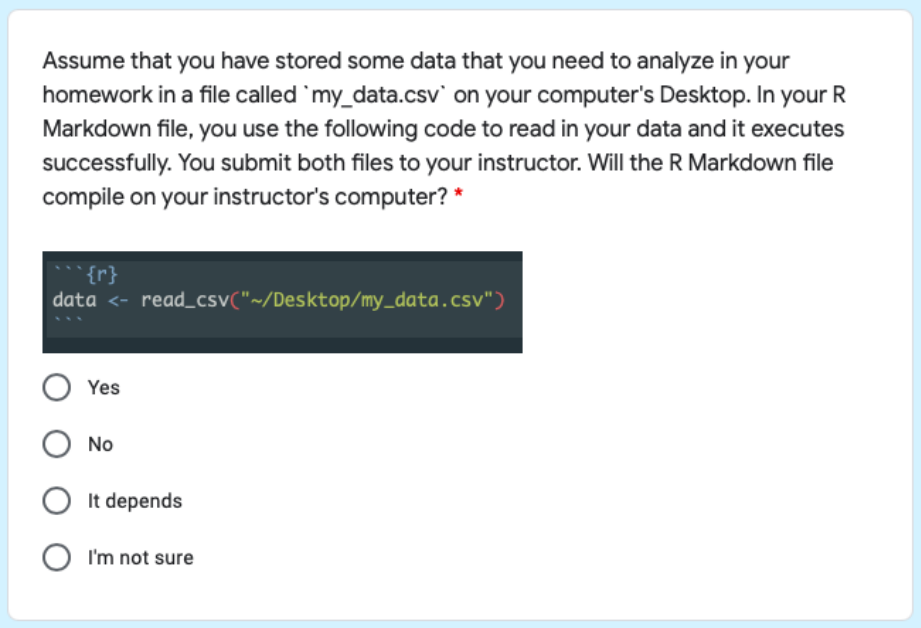
Not **abertin@smith.edu**? Switch account

Question 1 (Projects)



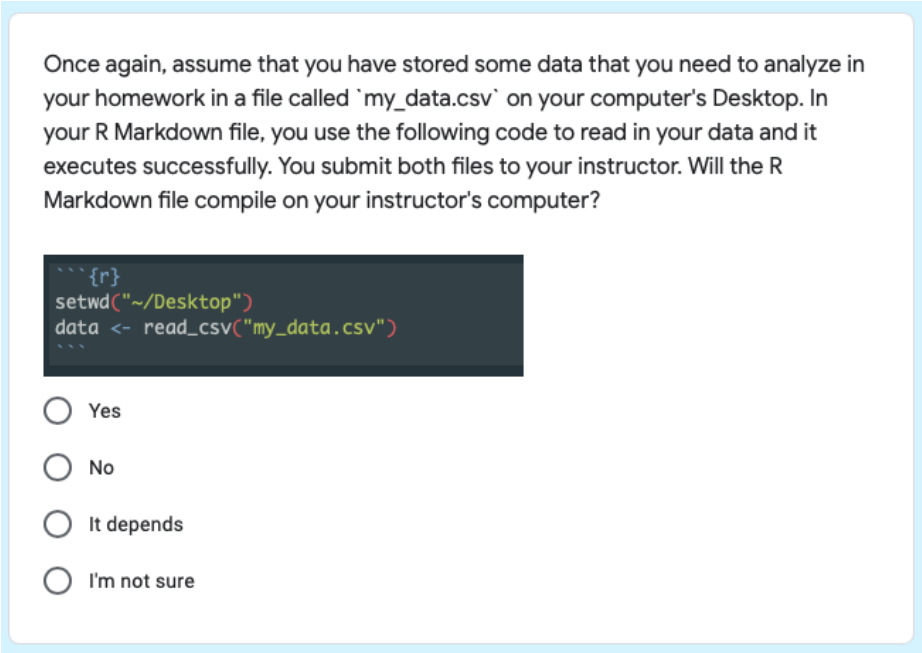
Correct Answers: 1) They make it easy to share all of your code/files with other people, 2) They help keep all your files in one place on your computer, separated from other unrelated files, 3) They allow you to connect to version controls like GitHub.

Question 2 (Projects)



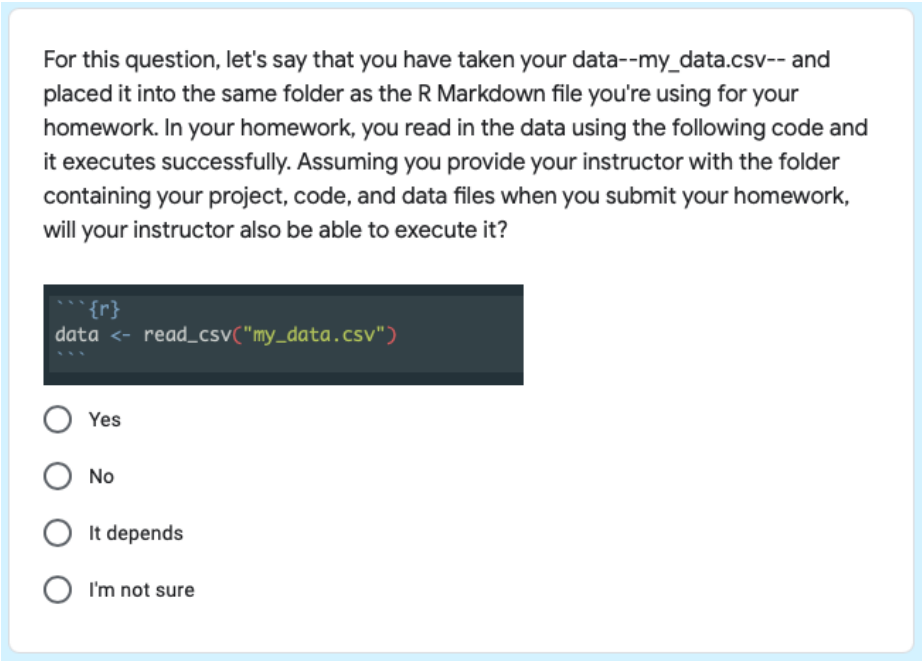
Correct Answer: "It depends"

Question 3 (Projects)



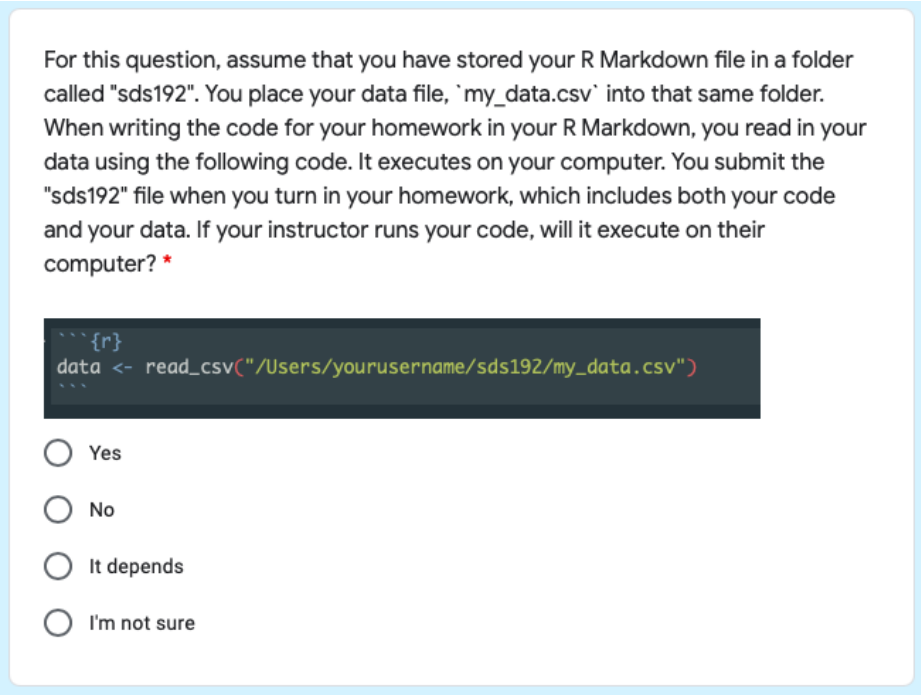
Correct Answer: "It depends"

Question 4 (Projects)



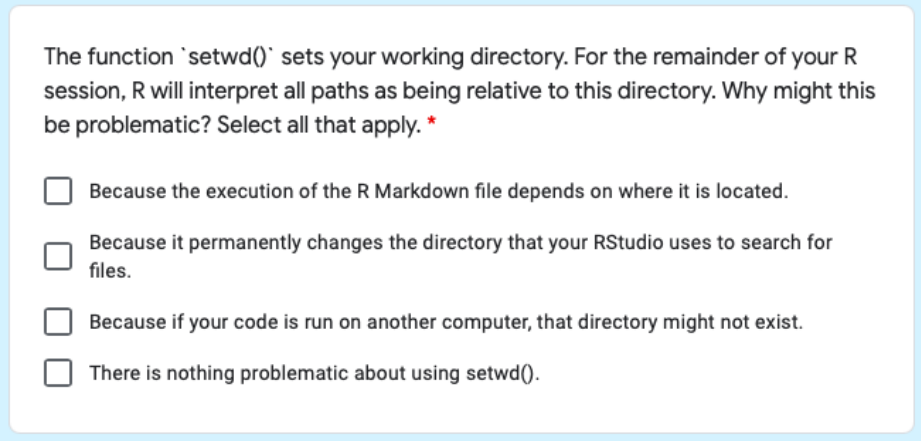
Correct Answer: "Yes"

Question 5 (Projects)



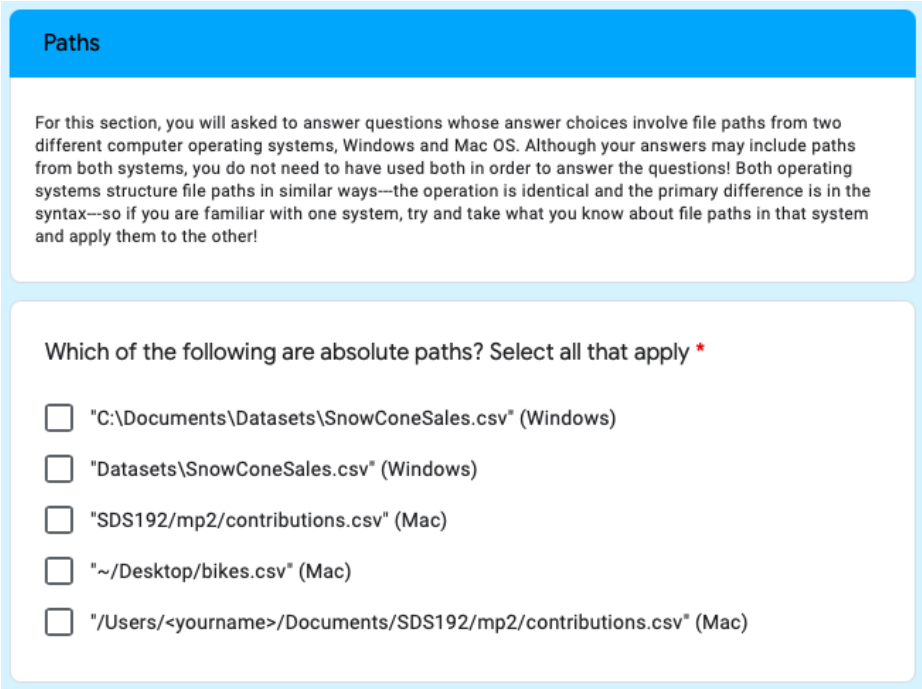
Correct Answer: "No"

Question 6 (Projects)



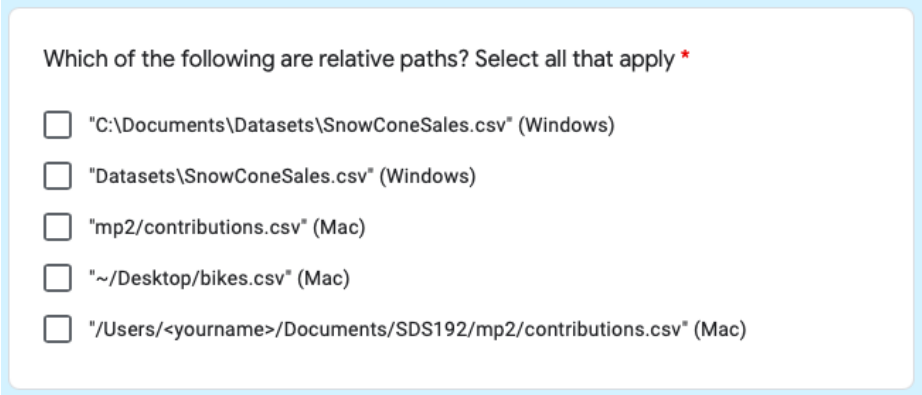
Correct Answers: 3) Because if your code is run on another computer, that directory might not exist. **Note**: *1) Because the execution of the R Markdown file depends on where it is located* is sort of ambiguous. Because of this, in scoring, it was considered to be a no change (0) answer if it was selected. Students would not gain or lose points from choosing that answer.

Question 7 (Paths)



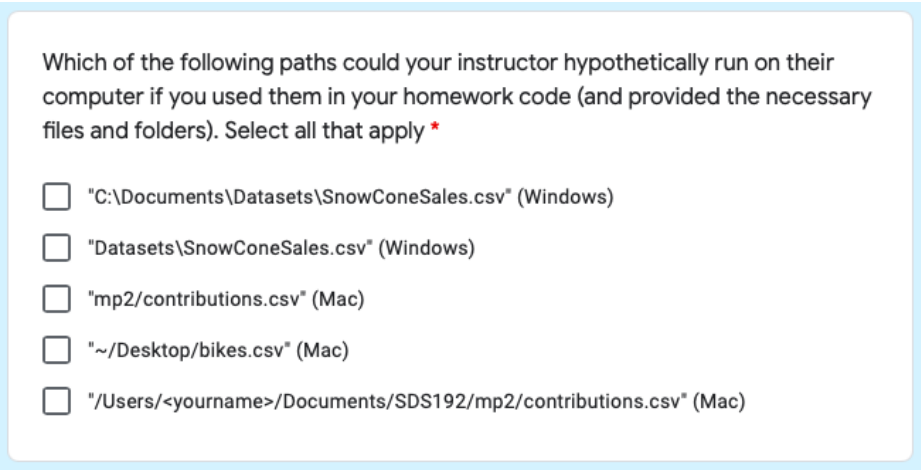
Correct Answers: 1) `"C:\Documents\Datasets\SnowConeSales.csv"` (Windows), 4) `"~/Desktop/bikes.csv"` (Mac), 5) `"/Users/<yourname>/Documents/SDS192/mp2/contributions.csv"` (Mac).

Question 8 (Paths)



Correct Answers: 2) `"Datasets\SnowConeSales.csv"` (Windows), 3) `"mp2/contributions.csv"` (Mac).

Question 9 (Paths)



Correct Answers: 2) `"Datasets\SnowConeSales.csv"` (Windows), 3) `"mp2/contributions.csv"` (Mac). **Note**: 1) and 4) might also be considered plausible, as they could theoretically work under some conditions, though after some discussion, Professor Baumer and I decided that they would be marked incorrect due to the wording of the question, which specifies that you provide the files and folders.

B.2 Sample Scoring Example

Below, we will consider a student's sample answers to this reproducibility test and how they will be scored. As discussed in Chapter 3, the following scoring method was used:

- For each question, students would start at a score of zero. Based on their answers, points would then be added, subtracted, or stay constant.
- On select-all-that apply questions, the following scoring rules were used. This meant that students who answered some things correctly, but other things incorrectly, could still receive negative or neutral scores, based on the overall analysis of how correct they were:
  - Each correct box that was checked –> +1 Point
  - Each incorrect box that was checked –> -1 Point
  - Each box not checked but should have been –> No change (0)
- On multiple choice questions, the scoring rules were slightly different. This was due to the inclusion of an option for "I'm not sure," which was included as a way to discourage random guessing and ensure that students were answering based on their actual knowledge.
  - If the correct answer was selected –> +1 Point
  - If the incorrect answer was selected –> -1 Point
  - If the student expressed uncertainty ("I'm not sure") –> No change (0)

Prior to scoring question 1, the total score of the student is set at 0, as is their projects questions score and paths questions score.

Question 1 (Projects)

A student has selected all of the following options:

- They make it easy to share all of your code/files with other people (CORRECT)
- They help keep all your files in one place on your computer, separated from other unrelated files (CORRECT)
- They allow you to connect to version control tools like GitHub (CORRECT)
- They speed up how fast your R session loads/saves (NOT CORRECT)
- They take up less space in your hard drive than when storing all of your files in one place (NOT CORRECT)

This student will receive [+1] for each of the three correct answers, and [-1] for each of the two incorrect ones, for a total of **[+1]**, for a new total score of 1. Since this question was about R projects, the student will also increase their `Projects Score` from 0 to 1.

Question Score: 1 *Projects Score*: 1 *Total Score*: 1

Question 2 (Projects)

The student has selected the following answer:

- Yes (NOT CORRECT) — the correct answer is "It Depends"

The student will receive [-1] for this answer, since it is incorrect. This will be added to their previous projects score and total score, resulting in the following updated scores:

Question Score: -1 *Projects Score*: 0 *Total Score*: 0

Question 3 (Projects)

The student has selected the following answer:

- Yes (NOT CORRECT) — the correct answer is "It Depends"

The student will receive [-1] for this answer, since it is incorrect. This will be added to their previous projects score and total score, resulting in the following updated scores:

Question Score: -1 *Projects Score*: -1 *Total Score*: -1

Question 4 (Projects)

The student has selected the following answer:

- Yes (CORRECT)

The student will receive [+1] for this answer, since it is correct. This will be added to their previous projects score and total score, resulting in the following updated scores:

Question Score: 1 *Projects Score*: 0 *Total Score*: 0

Question 5 (Projects)

The student has selected the following answer:

- No (CORRECT)

The student will receive [+1] for this answer, since it is correct. This will be added to their previous projects score and total score, resulting in the following updated scores:

Question Score: 1 *Projects Score*: 1 *Total Score*: 1

Question 6 (Projects)

The student has clicked one of the boxes:

- Because if your code is run on another computer, that directory might not exist (CORRECT)

The student will receive [+1] for this answer, since the box they chose was correct. Their choice not to select any other boxes will not affect their score. This will be added to their previous projects score and total score, resulting in the following updated scores:

Question Score: 1 *Projects Score*: 2 *Total Score*: 2

At the end of the section on projects, the student has gained a projects score of 2 and is at a total score of 2 as well.

Question 7 (Paths)

The student has clicked two of the boxes:

- `"C:\Documents\Datasets\SnowConeSales.csv"` (Windows) (CORRECT)
- `"~/Desktop/bikes.csv"` (Mac) (CORRECT)

The student will receive [+2] for this answer, since both of the boxes they chose was correct. Their choice not to select any other boxes will not affect their score. This will be added to their their paths score (which started at zero) and total score, resulting in the following updated scores:

Question Score: 2 *Paths Score*: 2 *Total Score*: 4

Question 8 (Paths)

The student has clicked three of the boxes:

- `"Datasets\SnowConeSales.csv"` (Windows) (CORRECT)
- `"mp2/contributions.csv"` (Mac) (CORRECT)
- `"/Users/<yourname>/Documents/SDS192/mp2/contributions.csv"` (Mac) (NOT CORRECT)

The student will receive [+1] for this answer, [+2] for the correct boxes and [-1] for the incorrect one. Their choice not to select any other boxes will not affect their score. This will be added to their their paths score and total score, resulting in the following updated scores:

Question Score: 1 *Paths Score*: 3 *Total Score*: 5

Question 9 (Paths)

The student has clicked two of the boxes:

- `"mp2/contributions.csv"` (CORRECT)
- `"/Users/<yourname>/Documents/SDS192/mp2/contributions.csv"` (Mac) (NOT CORRECT)

The student will receive [0] for this answer, [+1] for the correct box and [-1] for the incorrect one. Their choice not to select any other boxes will not affect their score. This will be added to their their paths score and total score, resulting in the following updated scores:

Question Score: 0 *Paths Score*: 3 *Total Score*: 5

At the end of the test, they will be left with a Projects Score of 2, a Paths Score of 3, and a Total Score of 5.

References

10 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

Arafat, O., & Riehle, D. (2009). The commenting practice of open source. In *Proceedings of the 24th ACM SIGPLAN conference companion on object oriented programming systems languages and applications* (pp. 857–864).

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*.

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`

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`

Karpicke, J. D., & Roediger, H. L. (2008). The critical importance of retrieval for learning. *Science*, *319*(5865), 966–968.

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`

Metskas, L. A., Kulp, M., & Scordilis, S. P. (2010). Gender dimorphism in the exercise-naive murine skeletal muscle proteome. *Cellular & Molecular Biology Letters*, *15*(3), 507–516.

National Institutes of Health. (2014). Principles and guidelines for reporting 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`

Prana, G. A. A., Treude, C., Thung, F., Atapattu, T., & Lo, D. (2019). Categorizing the content of GitHub README files. *Empirical Software Engineering*, *24*(3), 1296–1327.

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 effectiveness 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.

Wickham, H. (2020). The tidyverse style guide. Retrieved from https://style.tidyverse.org

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.