Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Creating optimal conditions for reproducible data analysis in R with 'fertile'†

Audrey M. Bertin[1] | Benjamin S. Baumer[2]

[1]Program in Statistical and Data Sciences, Smith College, Northampton, MA

[2]Associate Professor of Statistical and Data Sciences, Smith College, Northampton, MA

Correspondence
Audrey Bertin, Program in Statistical and Data Sciences, Smith College, Northampton, MA. Email: abertin@smith.edu

Summary

The advancement of scientific knowledge increasingly depends on ensuring that data-driven research is reproducible: that two people with the same data obtain the same results. However, while the necessity of reproducibility is clear, there are significant behavioral and technical challenges that impede its widespread implementation and no clear consensus on standards of what constitutes reproducibility in published research. We present fertile, an R package that focuses on a series of common mistakes programmers make while conducting data science projects in R, primarily through the RStudio integrated development environment. fertile operates in two modes: proactively, to prevent reproducibility mistakes from happening in the first place, and retroactively, analyzing code that is already written for potential problems. Furthermore, fertile is designed to educate users on why their mistakes are problematic and how to fix them.

KEYWORDS:
Quality control, Statistical computing, Statistical process control, Teaching Statistics

1 Introduction

Data-based research is considered fully reproducible when the requisite code and data files produce identical results when run by another analyst (Patil, Peng, & Leek 2019). As research is becoming increasingly data-driven, and because knowledge can be shared worldwide so rapidly, reproducibility is critical to the advancement of scientific knowledge. Academics around the world have recognized this, and publications and discussions addressing reproducibility appear to have increased in the last several years (Eisner 2018; Fidler & Wilcox 2018; Gosselin 2020; McArthur 2019; Wallach, Boyack, & Ioannidis 2018).

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

However, while the necessity of reproducibility is clear, there are significant behavioral and technical challenges that impede its widespread implementation and no clear consensus on standards of what constitutes reproducibility in published research (Peng 2009). Not only are the components of reproducible research up for discussion (e.g., need the software be open source?), but the corresponding recommendations for ensuring reproducibility also vary (e.g., should raw and processed data files be in separate directories?). Existing attempts to address reproducibility in data science are often either too generalized—resulting in shallow and vague recommendations that are challenging to implement—or too specific—approaching one aspect of reproducibility well but doing so in a highly technical way that fails to capture the bigger picture and creates challenges for inexperienced users.

†The authors gratefully acknowledge contributions from Hadley Wickham, Jenny Bryan, Greg Wilson, Edgar Ruiz, and other members of the **tidyverse** team.

In order to address these challenges, we present **fertile**(Baumer & Bertin 2020), a low barrier-to-entry package focusing on common reproducibility mistakes programmers make while conducting data science research in R (R Core Team 2020).

2 Related Work

2.1 Literature Review

Publications on reproducibility can be found in all areas of scientific research. However, as Goodman, Fanelli, and Ioannidis (2016) argue, the language and conceptual framework of research reproducibility ~~varies~~ vary significantly across the sciences, and there are no clear standards on reproducibility agreed upon by the scientific community as a whole. We consider recommendations from a variety of fields and determine the key aspects of reproducibility faced by scientists in different disciplines.

Kitzes, Turek, and Deniz (2017) present a collection of case studies on reproducibility practices from across the data-intensive sciences, illustrating a variety of recommendations and techniques for achieving reproducibility. Although their work does not come to a consensus on the exact standards of reproducibility that should be followed, several common trends and principles emerge from their case studies: 1) use clear separation, labeling, and documentation, 2) automate processes when possible, and 3) design the data analysis workflow as a sequence of small steps glued together, with outputs from one step serving as inputs into the next. This is a common suggestion within the computing community, originating as part of the Unix philosophy (Gancarz 2003).

Cooper et al. (2017) focus on data analysis in R and identify a similar list of important reproducibility components, reinforcing the need for clearly labeled, well-documented, and well-separated files. In addition, they recommend publishing a list of dependencies and using version control. 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, and include a *README*. Wilson et al. (2017) argue that, to follow good practice, an analysis must be located in a well-organized directory, be supplemented by a file containing information about the project (i.e. a README), and a provide an explicit list of dependencies. Wilson et al. (2014) emphasize communication, recommending that code be human-readable and consistently styled for ease of understanding once shared.

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

When considered in combination, these sources provide a well-rounded picture of the components important to research reproducibility. Using this literature as a guideline, we identify several key features of reproducible work. These recommendations are a matter of opinion—due to the lack of agreement on which components of reproducibility are most important, we select those that are mentioned most often, as well as some that are mentioned less but that we view as important.

1. A well-designed file structure:
    - Separate folders for different file types.
    - No extraneous files.
    - Minimal clutter.
2. Good documentation:
    - Files are clearly named, preferably in a way where the order in which they should be run is clear.
    - A README is present.
    - Dependencies are noted.
    - Code files contain descriptive comments.
3. Reproducible file paths:
    - No absolute paths, or paths leading to locations outside of a project's directory, are used in code—only portable (relative) paths.
4. Randomness is accounted for:
    - If randomness is used in code, a seed must also be set.
5. Readable, styled code:
    - Code should be written in a coherent style. Code that conforms to a style guide or is written in a consistent dialect is easier to read (Hermans & Aldewereld 2017). We believe that the **tidyverse** provides the most accessible dialect of R.

Much of the available literature focuses on file structure, organization, and naming, and **fertile**'s features are consistent with this. Marwick, Boettiger, and Mullen (2018) provide the framework for file structure that **fertile** is based on: a structure similar to that of an R package (R-Core-Team 2020; Wickham 2015), with an *R* folder, as well as *data*, *data-raw*, *inst*, and *vignettes*.

2.2 R Packages and Other Software

Much of the work discussed in Section 2.1 is highly generalized, written to be applicable to users working with a variety of statistical software programs. Because all statistical software programs operate differently, these recommendations are inherently vague and difficult to implement, particularly to new analysts who are relatively unfamiliar with their software. Focused attempts to address reproducibility in specific certain software programs are more likely to be successful. We focus on R, due to its open-source nature, accessibility, and popularity as a tool for statistical analysis.

A small body of R packages focuses on research reproducibility. **rrtools** (Marwick 2019) addresses some of the issues discussed in Marwick et al. (2018) by creating a basic R package structure for a data analysis project and implementing a basic *testthat::check*() functionality. 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. **workflowr**'s (J. 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. **checkers** (Ross, DeCicco, & Randhawa 2018) allows you to create custom checks that examine different aspects of reproducibility. **packrat** (Ushey, McPherson, Cheng, Atkins, & Allaire 2018) is focused on dependencies, creating a packaged folder containing a project as well as all of its dependencies so that projects dependent on lesser-used packages can be easily shared across computers. **drake** (OpenSci 2020) analyzes workflows, skips steps where results are up to date, 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.

Many of these packages are narrow, with each effectively addressing a small component of reproducibility: file structure, modularization of code, version control, etc. These packages 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 near the beginning of their R journey, nor particularly useful to those looking for quick and easy reproducibility checks. A more effective way of realizing widespread reproducibility is to make the process for doing so simple enough that it takes little to no conscious effort to implement. You want users to "fall into a hole"[1] of good practice.

Continuous integration tools 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.

However, while these tools can be useful, they are generalized so as to be useful to the widest audience. As a result, their checks are not designed to be R-specific, which makes them sub-optimal for users looking to address reproducibility issues involving features specific to the R programming language, such as package installation and seed setting.

2.3 Our contribution

fertile attempts to address these gaps in existing software by providing a simple, easy-to-learn reproducibility package that, rather than focusing intensely on a specific area (such as file structure), provides some information about a wide variety of aspects influencing reproducibility. For example, **fertile** addresses file structure in part but also considers the quality of project documentation, the use of file paths, and the behavior of random number generation. **fertile** is flexible, ~~offering benefits~~ providing reports on reproducibility to users at any stage in the data analysis workflow, and provides R-specific features, such as tools to manage package dependencies, which address certain aspects of reproducibility that can be missed by external project development software.

fertile is designed to be used on data analyses organized as R Projects (i.e. directories containing an *.Rproj* file). Once an R Project is created, **fertile** provides ~~benefits~~ the user with feedback on their adherence to reproducibility standards throughout the data analysis process, both during development as well as after the fact. **fertile** achieves this by operating in two modes: proactively, to prevent reproducibility mistakes from happening in the first place, and retroactively, analyzing code that has already been written for potential problems.

[1] We paraphrase Hadley Wickham.

3 Methods

3.1 Proactive Use

Proactively, the package identifies potential mistakes as they are made by the user and outputs an informative message as well as a recommended solution. For example, **fertile** ~~catches~~ throws an error when a user passes a potentially problematic file path—such as an absolute path, or a path that points to a location outside of the project directory—to a variety of common input/output functions operating on many different file types.

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

## [1] TRUE

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')

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')
```

Users who want to override this behavior and execute their function anyway are provided with a piece of code to execute that will achieve this for them.

fertile is even more aggressive with functions (like *setwd*()) that are almost certain to break reproducibility no matter the type of path passed to them. ~~, causing them to throw errors that prevent their execution and providing recommendations for better alternatives~~. When these functions are called, users are not presented with an option to override the resulting error. Instead, the only way to achieve the desired action is for the user to execute an alternative function that produces the same behavior but in a reproducible way.

```
setwd("~/Desktop")

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

3.2 Retroactive Use

Retroactively, **fertile** analyzes potential obstacles to reproducibility in an RStudio Project (i.e., a directory that contains an *.Rproj* file). The package considers several different aspects of the project which may influence reproducibility, including the directory structure, file paths, and whether randomness is used thoughtfully. The end products of these analyses are reproducibility reports summarizing a project's adherence to reproducibility standards and recommending remedies for where the project falls short. For example, **fertile** might identify the use of randomness in code and recommend setting a seed if one is not present.

Users can access the majority of **fertile**'s retroactive features through two primary functions, *proj_check*() and *proj_analyze*().

The *proj_check*() function runs sixteen different reproducibility tests, 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 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 in Table 1:

Subsets of the sixteen tests can be invoked using the **tidyselect** helper functions (Henry & Wickham 2020) in combination with the more limited *proj_check_some*() function. These helper functions, which include *starts_with*(), *ends_with*(), *contains*(), and *matches*(), allow users to check for matches between a string or regular expression and a list of variables–in this case, searching for checks from the list provided by *list_checks*() whose names match a certain condition (e.g. ending with "root").

TABLE 1 Table illustrating the list of check available in fertile.

Check	Functionality
has_proj_root()	Checks to make sure a single .Rproj file is found in the root directory of the project.
has_no_nested_proj_root()	Checks to make sure there are no nested .Rproj files in subfolders of the project.
has_clear_build_chain()	Looks for indication of a clear order in which to run R scripts.
has_tidy_[media/images/code/raw_data/data/scripts]()	There are six different 'has_tidy_...' functions. Each checks to make sure that no files of its given type (images, code, data, etc.) are located in the root directory of the project
has_only_used_files()	Ensures that all files located in the project directory are being referenced or created by other files in the project.
has_readme()	Checks to make sure there is a README file in the root directory.
has_well_commented_code()	Checks to ensure that all code files contain an adequate number of comments (>10% of lines).
has_no_absolute_paths()	Checks to make sure that no absolute paths are referenced in code.
has_only_portable_paths()	Checks to make sure that no paths in code lead outside the project directory.
has_no_randomness()	Checks to make sure code does not contain uncontrolled randomness.
has_no_lint()	Examines code to determine whether it conforms to 'tidy' style guide recommendations.

```
proj_dir <- "project_miceps"
```

```
proj_check_some(proj_dir, contains("paths"))

## - Compiling... -------------------------- fertile 0.0.0.9028 -
## - Rendering R scripts... --------------------- fertile 0.0.0.9028 -
## Error in eval_tidy(xs[[j]], mask): object '.Random.seed' not found
## - Running reproducibility checks ----------------- fertile 0.0.0.9028 -
## Error: arrange() failed at implicit mutate() step.
## x Could not create a temporary column for `..1`.
## i `..1` is `timestamp`.
```

Each test can also be run individually by calling the function matching its check name.

The *proj_analyze*() function creates a report documenting the structure of a data analysis project. This report contains information about all packages referenced in code, the files present in the directory and their types, suggestions for moving files to create a more organized structure, and a list of reproducibility-breaking file paths used in code.

```
proj_analyze(proj_dir)

## Error: arrange() failed at implicit mutate() step.
## x Could not create a temporary column for `..1`.
## i `..1` is `timestamp`.
```

Both the proactive and retroactive features available in **fertile** are enabled automatically upon the loading of the package. To prevent users from accidentally enabling or disabling features, **fertile** does not provide any option to control which features are active. However, those looking to only use the proactive features can do so by refraining from using the retroactive-specific functions while those looking to only use the retroactive features can wait to load **fertile** until they are ready to produce a report on their project.

3.3 Logging

fertile also contains logging functionality, which records commands run in the console that have the potential to affect reproducibility, enabling users to look at their past history at any time. The package focuses mostly on package loading and file opening, noting which function was used,

the path or package it referenced, and the timestamp at which that event happened. Users can access the log recording their commands at any time via the *log_report*() function:

```
log_report()

## # A tibble: 3 x 4
##   path         path_abs                                  func     timestamp
##   <chr>        <chr>                                     <chr>    <dttm>
## 1 package:pur~ <NA>                                      base::l~ 2020-10-09 23:56:28
## 2 package:for~ <NA>                                      base::l~ 2020-10-09 23:56:28
## 3 project_mic~ /Users/audreybertin/Documents/ferti~ readr::~ 2020-10-09 23:56:28
```

The log, if not managed, can grow very long over time. For users who do not desire such functionality, *log_clear*() provides a way to erase the log and start over.

3.4 How It Works

Much of the functionality in **fertile** is achieved by writing shims. In computer programming generally, a shim is a function that intercepts a user's call, changes the arguments passed to it, and/or modifies the execution of the function in some way. In the case of **fertile** specifically, shims are used to intercept the user's commands and perform various logging and path-checking tasks before executing the desired function. Our process is:

1. Identify an R function that is likely to be involved in operations that may break reproducibility. Popular functions associated with only one package (e.g., *read_csv*() from **readr**) are ideal candidates.
2. Create a function in **fertile** with the same name that takes the same arguments ~~(and always the dots ...)~~.
3. Write this new function so that it: a) captures any arguments, b) logs the name of the function called, c) performs any checks on these arguments, and d) calls the original function with the original arguments. Except where warranted, the execution looks the same to the user as if they were calling the original function. Since **fertile** only needs access to the file path argument, "file" is the only declared argument for most shims. The other arguments are managed using the dots (...), which capture and save them, allowing them to eventually be passed back to the original shimmed function (e.g. *readr*::*read_csv*()).

Most shims are quite simple and look something like what is shown below for *read_csv*().

```
fertile::read_csv

## function(file, ...) {
##   if (interactive_log_on()) {
##     log_push(file, "readr::read_csv")
##     check_path_safe(file, ... = "readr::read_csv")
##     readr::read_csv(file, ...)
##   }
## }
## <bytecode: 0x7fe67a348478>
## <environment: namespace:fertile>
```

fertile shims many common functions, including those that read in a variety of data types, write data, and load packages. This works both proactively and retroactively, as the shimmed functions written in **fertile** are activated both when the user is coding interactively and when a file containing code is rendered.

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*(), we detach **fertile** from the search path, load the requested package, and then re-attach **fertile**. This ensures that when a user executes a command, R will check **fertile** for a matching function before considering other packages. While it is possible that this shifty behavior could lead to unintended consequences, our goal is to catch a good deal of problems before they become problematic. Users can easily disable **fertile** by detaching it, or not loading it in the first place.

3.5 Utility Functions

fertile also provides several useful utility functions that may assist with the process of data analysis.

3.5.1 File Paths

The *check_path*() function analyzes a vector of paths (or a single path) to determine whether there are any absolute paths or paths that lead outside the project directory.

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

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

# 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.

# Path outside the directory
check_path("../fertile.Rmd")

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

3.5.2 File Types

There are several functions that can be used to check the type of a file:

```
is_data_file(fs::path(proj_dir, "mice.csv"))

## [1] TRUE

is_image_file(fs::path(proj_dir, "proteins_v_time.png"))

## [1] TRUE

is_text_file(fs::path(proj_dir, "README.md"))

## [1] TRUE

is_r_file(fs::path(proj_dir, "analysis.Rmd"))

## [1] TRUE
```

3.5.3 Temporary Directories

The *sandbox*() function allows the user to make a copy of their project in a temporary directory. This can be useful for ensuring that projects run properly when access to the local file system is removed.

```
proj_dir

## [1] "project_miceps"

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

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

temp_dir <- sandbox(proj_dir)
temp_dir

## /var/folders/v6/f62qz88s0sd5n3yqw9d8sb300000gn/T/RtmpisOoIV/project_miceps

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

## /var/folders/v6/f62qz88s0sd5n3yqw9d8sb300000gn/T/RtmpisOoIV/project_miceps/Blot_data_updated.csv
## /var/folders/v6/f62qz88s0sd5n3yqw9d8sb300000gn/T/RtmpisOoIV/project_miceps/CS_data_redone.csv
## /var/folders/v6/f62qz88s0sd5n3yqw9d8sb300000gn/T/RtmpisOoIV/project_miceps/Estrogen_Receptors.docx
```

3.5.4 Managing Project Dependencies

One of the challenges with ensuring that work is reproducible is the issue of dependencies. Many data analysis projects reference a variety of R packages in their code. When such projects are shared with other users who may not have the required packages downloaded, it can cause errors that prevent the project from running properly.

The *proj_pkg_script*() function assists with this issue by making it simple and fast to download dependencies. 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.

```
install_script <- proj_pkg_script(proj_dir)
```

The package installation script for project miceps looks as follows:

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

fertile also keeps track of dependencies in additional detail. 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.

Users can access this file via the *proj_dependency_report*() function. Its output appears as a new window in RStudio with contents that look like the text below:

```
##  The R project located at '/Users/audreybertin/Documents/fertile-paper2/STAT-paper-template
##  /APA/project_miceps' was last run in the following software environment:

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

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

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

##  attached base packages:
```

```
##  stats     graphics  grDevices utils     datasets  methods   base

##  other attached packages:
##  stargazer_5.2.2    skimr_2.1.2        purrr_0.3.4
##  ggplot2_3.3.2      tidyr_1.1.2        readr_1.3.1        dplyr_1.0.2
```

3.6 Sample Use Cases

fertile's simplicity enables users of any background to take advantage of its features and its big-picture design gives **fertile** the potential to provide benefits across a variety of disciplines.

For example, professors could integrate **fertile** into their data science curricula, giving students an understanding and awareness of reproducibility early in their careers that can positively impact the reproducibility of their future work. It could also be used by experienced analysts working collaboratively who are looking to promote a smoother exchange of feedback and ideas. Journal reviewers may also find the package beneficial, allowing them to gain a fast overview of whether paper submissions meet reproducibility guidelines.

The sample use cases in this section consider **fertile**'s applicability to some of these scenarios in detail.

3.6.1 Introductory Data Science Student

Susan is taking an introductory data science course. This is her first time learning how to code and she has not yet been exposed to ideas of research reproducibility. Her professor has assigned a data analysis project that must be completed in R Markdown. The project requires her to read in a data file located on her computer and use it to produce a graph.

She reads in the data, makes the graph, and knits her *.Rmd* file. It compiles successfully, so she submits the assignment. The next day, she receives an email from her professor saying that her assignment failed to compile and that she needs to make changes and try again. Susan does not understand why it did not work on the professor's computer when it did on her own. The professor recommends that she install **fertile** and run *proj_check*() on her assignment. She does this and gets a message informing her that she used an absolute path to open her dataset when she should have use a relative path instead. She looks up what this means and then uses the new information to update her assignment. Her second submission compiles successfuly.

On future projects, she always loads and runs **fertile** before submitting.

3.6.2 Experienced R User

Emma is a post-doc with several years of R experience. She is familiar with some basic rules of reproducibility—file paths should always be relative and randomness should always be associated with a seed—but has never needed to pass any sort of reproducibility check before because her professors never emphasized that.

She has just finished a research project and is looking to submit her work to a journal. When researching the journal to which she is interested in submitting, she discovers that it has high standards for research reproducibility and a dedicated editor focusing on that aspect of submission. She goes online and finds the journal's guidelines for reproducibility. They are more complete than any guidelines to which she has previously been required to conform. In addition to notes about file paths and randomness, the journal requires a clean, well-organized folder structure, broken down by file category and stripped of files that do not serve a purpose. In order to be approved, submissions must also have a clear build chain and an informative *README* file.

Unsure of the best way to achieve this structure, Emma goes online to find help. In her search, she comes across **fertile**. She downloads the package, and in only a handful of commands, she identifies and removes excess files in her directory and automatically organizes her files into a structure reminiscent of an R package. She now meets the guidelines for the journal and can submit her research.

4 Experimental Testing

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

To test **fertile**'s effectiveness, we began an initial randomized control trial of the package on an introductory undergraduate data science course at Smith College in Spring 2020 [2].

The experiment was structured as follows:

[2] This study was approved by Smith College IRB, Protocol #19-032

- Students are given a form at the start of the semester asking whether they consent to participate in a study on data science education. In order to successfully consent, they must provide their system username, collected through the command *Sys.getenv("LOGNAME")*. To maintain privacy the results are then transformed into a hexadecimal string via the *md5*() hashing function.
- These hexadecimal strings are then randomly assigned into equally sized groups, one experimental group that receives the features of **fertile** and one group that receives a control.
- The students are then asked to download a package called **sds192** (the course number and prefix), which was created for the purpose of this trial. It leverages an *.onAttach*() function to scan the R environment and collect the username of the user who is loading the package and run it through the same hashing algorithm as used previously. It then identifies whether that user belongs to the experimental or the control group. Depending on the group they are in, they receive a different version of the package.
- The experimental group receives the basic **sds192** package, which consists of some data sets and R Markdown templates necessary for completing homework assignments and projects in the class, but also has **fertile** installed and loaded silently in the background. The package's proactive features are enabled, and therefore users will receive warning messages when they use absolute or non-portable paths or attempt to change their working directory. The control group receives only the basic **sds192** package, including its data sets and R Markdown templates. All students from both groups then use their version of the package throughout the semester on a variety of projects.
- Both groups are given a short quiz on different components of reproducibility that are intended to be taught by **fertile** at both the beginning and end of the semester. Their scores are then compared to see whether one group learned more than the other group or whether their scores were essentially equivalent. Additionally, for every homework assignment submitted, the professor takes note of whether or not the project compiles successfully.

Based on the results, we hope to determine whether **fertile** was successful at achieving its intended goals. A lack of notable difference between the *experimental* and *control* groups in terms of the number of code-related questions asked throughout the semester would indicate that **fertile** achieved its goal of simplicity. A higher average for the *experimental* group in terms of the number of homework assignments that compiled successfully would indicate that **fertile** was successful in increasing reproducibility. A greater increase over the semester in the reproducibility quiz scores for students in the *experimental* group compared with the *control* group would indicate that **fertile** achieved its goal of educating users on reproducibility. Success according to these metrics would provide evidence showing **fertile**'s benefit as tool to help educators introduce reproducibility concepts in the classroom.

Unfortunately, we were unable to complete the analysis as intended as the trial had to be postponed after the COVID-19 pandemic significantly altered the experimental conditions at the midpoint of testing. Although the experiment was unsuccessful in its first attempt, ~~we hope to run the same trial again and gather data on fertile's effectiveness~~ conditions at Smith College have now stabilized and we are currently running a second trial in the new online-only classroom environment.

5 Conclusion

fertile is an R package that lowers barriers to reproducible data analysis projects in R, providing a wide array of checks and suggestions addressing many different aspects of project reproducibility, including file organization, file path usage, documentation, and dependencies. **fertile** is meant to be educational, providing informative error messages that indicate why users' mistakes are problematic and sharing recommendations on how to fix them. The package is designed in this way so as to promote a greater understanding of reproducibility concepts in its users, with the goal of increasing the overall awareness and understanding of reproducibility in the R community.

The package has very low barriers to entry, making it accessible to users with various levels of background knowledge. Unlike many other R packages focused on reproducibility that are currently available, the features of **fertile** can be accessed almost effortlessly. Many of the retroactive features can be accessed in only two lines of code requiring minimal arguments and some of the proactive features can be accessed with no additional effort beyond loading the package. This, in combination with the fact that **fertile** does not focus on one specific area of reproducibility, instead covering (albeit in less detail) a wide variety of topics, means that **fertile** makes it easy for data analysts of all skill levels to quickly gain a better understanding of the reproducibility of the work.

In the moment, it often feels easiest to take a shortcut—to use an absolute path or change a working directory. However, when considering the long term path of a project, spending the extra time to improve reproducibility is worthwhile. **fertile**'s user-friendly features can help data analysts avoid these harmful shortcuts with minimal effort.

5.1 Future Work

fertile, in its current version, addresses the vast majority of the aspects of reproducibility identified in Section 2.1 in some way. However, there are several areas where further development to extend the available features of the package would be beneficial. These include the following:

- ~~Expanding dependency management features to include R session information and package version numbers in addition to package names.~~

- ~~Expanding code and documentation style features to analyze whether code has been properly commented in addition to checking for a README and tidy code style~~.
- Adding *make*-like functionality that can analyze an R project structure and files and use this information to generate a Makefile. This Makefile would have information about target files and their prerequisites and would assist with making sure that re-running an analysis is done as quickly as possible by ensuring that only the necessary code and files that have been updated are run when rebuilding and re-running code.
- Adding a badge system to improve user experience and simplify the process of understanding how reproducible any given project is. Under this system, projects would receive badges from **fertile** based on the sets of reproducibility checks they pass. For example, projects that pass all of the checks on file paths would receive a badge on their project for "Reproducible File Paths" while those who pass checks involving file clutter and project structure might be awarded a badge for "Exellent File Organization."
- Implementing a method through which users can add their own shims to the package. This would allow users to modify **fertile** to fit the functions they use most often.

Data Availability Statement

The sample project *project_miceps* and package code associated with this paper can be found in the *R* and *tests* folders at `https://github.com/baumer-lab/fertile`.

References

Baker, M. (2016). 1,500 scientists lift the lid on reproducibility. *Nature News*, *533*(7604), 452. Retrieved from `https://www.nature.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970`

Baumer, B. S., & Bertin, A. M. (2020). *fertile: creating optimal conditions for reproducibility.* Retrieved from `https://github.com/baumer-lab/fertile` R package version 0.0.0.9027.

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` R package version 1.6.0.

Blischak, J. D., Carbonetto, P., & Stephens, M. (2019). Creating and sharing reproducible research code the workflowr way. *F1000Research*, *8*(1749). Retrieved from `https://doi.org/10.12688/f1000research.20843.1` doi: 10.12688/f1000research.20843.1

Broman, K. (2019). *initial steps toward reproducible research: organize your data and code.* Retrieved from `https://kbroman.org/steps2rr/pages/organize.html`

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.* 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. Retrieved from `http://www.sciencedirect.com/science/article/pii/S0022282817303334` doi: https://doi.org/10.1016/j.yjmcc.2017.10.009

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

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` R package version 1.0.4.

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. Retrieved from `https://stm.sciencemag.org/content/8/341/341ps12` doi: 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. Retrieved from `https://onlinelibrary.wiley.com/doi/abs/10.1002/bies.201900189` doi: 10.1002/bies.201900189

Henry, L., & Wickham, H. (2020). *tidyselect: Select from a set of strings.* Retrieved from `https://CRAN.R-project.org/package=tidyselect` R package version 1.1.0.

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

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`

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 R package version 0.1.0.

Marwick, B., Boettiger, C., & Mullen, L. (2018). Packaging data analytical work reproducibly using R (and friends). *The American Statistician*, *72*(1), 80–88. Retrieved from https://peerj.com/preprints/3192.pdf doi: 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. Retrieved from https://doi.org/10.1116/1.5093621 doi: 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 R package version 1.0.0.

OpenSci, R. (2020). *drake: A pipeline toolkit for reproducible computation at scale*. Retrieved from https://cran.r-project.org/package=drake R package version 7.11.0.

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

Patil, P., Peng, R. D., & Leek, J. T. (2019). A visual tool for defining reproducibility and replicability. *Nature human behaviour*, *3*(7), 650–652. Retrieved from https://www.nature.com/articles/s41562-019-0629-z

Peng, R. D. (2009, 07). Reproducible research and Biostatistics. *Biostatistics*, *10*(3), 405-408. Retrieved from https://doi.org/10.1093/biostatistics/kxp014 doi: 10.1093/biostatistics/kxp014

Popper, K. (2005). *The logic of scientific discovery*. New York, NY: Routledge.

R Core Team. (2020). R: A language and environment for statistical computing [Computer software manual]. Vienna, Austria. Retrieved from https://www.R-project.org/

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 R package version 0.1.0.

Ushey, K., McPherson, J., Cheng, J., Atkins, A., & Allaire, J. (2018). *packrat: A dependency management system for projects and their r package dependencies*. Retrieved from https://CRAN.R-project.org/package=packrat R package version 0.5.0.

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

Wickham, H. (2011). testthat: Get started with testing. *The R Journal*, *3*, 5–10. Retrieved from https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf

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

Wickham, H. (2019a). *testthat: Unit testing for R*. Retrieved from https://CRAN.R-project.org/package=testthat R package version 2.3.1.

Wickham, H. (2019b). *tidyverse: Easily install and load the 'tidyverse'*. Retrieved from https://CRAN.R-project.org/package=tidyverse R package version 1.3.0.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., … Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, *4*(43), 1686. Retrieved from https://tidyverse.tidyverse.org/articles/paper.html doi: 10.21105/joss.01686

Wickham, H., François, R., Henry, L., & Müller, K. (2019). *dplyr: A grammar of data manipulation*. Retrieved from https://CRAN.R-project.org/package=dplyr R package version 0.8.3.

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

Author Biography

Audrey M. Bertin is an undergraduate student at Smith College intending on completing a Bachelor of Arts in Statistical and Data Sciences and Public Policy in Spring 2021. Audrey is a recipient of Smith College's prestigious Zollman Scholarship and has participated in extensive research with faculty throughout her time at the college. Her research interests include data and science communication, data protection and privacy, algorithmic bias and injustice, and using data science for social good.

Benjamin S. Baumer is an associate professor in the Statistical and Data Sciences program at Smith College. Ben is a co-author of *The Sabermetric Revolution*, *Modern Data Science with R*, and the second edition of *Analyzing Baseball Data with R*. Ben has received the Waller Education Award from the ASA Section on Statistics and Data Science Education, the Significant Contributor Award from the ASA Section on Statistics in Sports, and the Contemporary Baseball Analysis Award from the Society for American Baseball Research. His research interests include sports analytics, data science, statistics and data science education, statistical computing, and network science.