Reproducibility made simple

Automating reproducible research workflows

Aaron Peikert

2020-07-08

Contents

Abstract

This is a short summary.

1 Theoretical Considerations

Claerbout & Karrenbach (1992) define reproducibility as the ability to gain the same results, from the same dataset. Conversely, they call a result replicable if one draws the same conclusion from a new dataset. This thesis concerns itself with the former, providing researchers with an accessible analysis workflow, that is virtually guaranteed to reproduce across time and devices.

The scientific community agrees that their work should be ideally reproducible. Indeed it may be hard to find a researcher who distrusts a result because it is reproducible; to the contrary, many argue it is "good scientific practice" to ensure what they consider reproducible ("Reducing our irreproducibility," 2013; Deutsche Forschungsgemeinschaft, 2019; Epskamp, 2019). Several reasons, practical and meta-scientific, justify this consensus of reproducibility as a minimal standard of Science.

Reproducibility makes researchers life more productive in two ways: The act of reproduction provides, at the most basic level, an opportunity for the researcher to spot errors. At the same time, other researchers may also benefit from reusing materials from an analysis they reproduced.

Beyond these two purely pragmatic reasons, reproduction is crucial, depending on the philosophical view of Science one subscribes to, because it allows independent validation and enables replication. Philosophers of Science characterise Science mainly as a shared method of determining whether or not a statement about the world is "true" (Andersen & Hepburn, 2016) or more broadly evaluating the statements verisimilitude (Gilbert, 1991; Meehl, 1990; Popper, 1962; Tichỳ, 1976). If this method is for experts to agree on the assumptions and deduce "truth", reproducibility is hardly necessary. On the other hand, it does gain importance if one induces facts by carefully observing the world. The decisive difference is that the former gains credibility by the authority of the experts, while the latter is trustworthy because anyone may verify it. Accepting induction as a scientific method hence hinges on the verifiability by others. Some have even argued that such democratisation of Science is what fueled the so-called scientific revolution (Heilbron, 2004, Scientific Revolution). The scientific revolution had the experiment as an agreed-upon method to observe reality, and a much later revolution provides statistical modelling (Rodgers, 2010) as a means to induction. This consensus about how to observe and how to induce gives modern scientific enterprises much of its credibility. Two reasons justify why we must assume reproducibility as a scientific standard if we accept induction as a scientific method: First, it enables independent verification of the process of induction, and second, it dramatically simplifies replication as a means to verify the induced truths.

However, neither the practical reasons that results might be less error-prone and more reusable nor the meta-scientific grounds that the process of induction and the induced facts are more straightforward to verify, if reproducible, derive strictly from the definition of reproducibility provided by Claerbout & Karrenbach (1992) given above. A simple thought experiment illustrates this shortcoming: Imagine a binary program being perfectly reproducible; hence upon the input of the same dataset, it fills a scientific manuscript with the same numbers at the right places. Furthermore, let us assume this hypothetical program may never hold if the dataset changes. Does the predicate "reproducible" in this situation reduce the number of mistakes or enables reuse? Unlikely. Or could one audit it and use it in replication? Hardly. This admittedly constructed case of a reproducible black box shows that we are not interested in reproducibility but rather in its side effects.

Spoiling its elegant simplicity, I extend the definition by Claerbout & Karrenbach (1992) to address this issue, by further demanding that reproducibility must facilitate replication. Hence, I would only call a result reproducible if the results remain unchanged if the data does, and it furthermore helps other researchers to replicate the results if they attempt to. With such a notion, the only valid cause of reproducibility is transparency. Only if it is clear how data relates to its results, both reproducibility and replication get promoted. Consequently, something is no longer either reproducible or not, but there are shades because a research product can promote replication to varying degrees. Note, that a scientific result can facilitate replication without anyone ever attempting to replicate it, e.g. by educating other researches about the analyses method, being openly accessible and providing reusable components.

Hence reproducibility has a technical side, which is ensuring the same results, and a non-technical side, which is facilitating understanding. The former relates to the practical advantages while the latter serves the metascientific purposes of reproducibility. An important caveat of the technical aspect is that generating the same results from the same data should always be possible regardless of time and machine. Following a reproducible analysis should be:

1. understandable by other researchers
2. transferable across machines
3. conserved through time.

This much more demanding standard of reproducibility is justified by two recent developments in the social sciences in general and psychology in particular: the emergence of a "replication crises" (Ioannidis, 2005) and the rise of "machine learning" (Jordan & Mitchell, 2015) as a scientific tool. Both trends link to the use of statistical modelling on which the social sciences became reliant for testing and

developing their theories (Gigerenzer et al., 2004; Meehl, 1978). It turns out that, if one fits the very same statistical model as published on newly gathered data, one fails more often to achieve the same results as published then one succeeds. (Open Science Collaboration, 2015).

Such failure to replicate findings that were believed to be robust has grown to a level that some social scientists call a crisis. They put forth various causes and remedies to this crisis. Most remedies share a common motif: transparency. Some call for Bayesian statistics (Maxwell et al., 2015), as it makes assumptions more explicit, or demand preregistration (Nosek et al., 2018) as a means to clarify how to analyse the data, beforehand and publicly. Others require the researchers to publish their data (Boulton et al., 2012). Similar calls for transparency, as a response to the replication crises, have formed the open science movement which stresses the necessity of six principles (Kraker et al., 2011):

- Open Access,
- Open Data,
- Open Source,
- Open Methodology,
- Open Peer Review and
- Open Educational Resources.

I argue that a research product resting on these pillars facilitates replication optimally and hence, it satisfies the highest standard of reproducibility. If everyone has access to a scientific product and its data along with the source code, everyone has the possibility of understanding the underlying methodology, which enables them to criticise the results and educate themselves. Having done so, they are in the best position for replication. Hence, any one's ability to reproduce such a result gives a tangible affirmation of its usefulness to the scientific community.

While reproducibility is no hurdle if one can perform the calculations needed with a pocket calculator, the more and more frequent use of computer-intensive methods renders such expectation questionable. The use of machine learning techniques, which has been once enabled by the computer taking over strenuous works, now impedes our quest for reproducibility. More massive amounts of more complicated computer code than ever before create room for errors and misunderstandings, leading the machine learning community to believe that they face a reproducibility crisis themselves (Hutson, 2018). Yet, I am far from calling for abstinence from machine learning, just because it complicates reproduction, but want to emphasise the need for solutions that allow anyone to reproduce even the most sophisticated analysis.

Peikert & Brandmaier (2019) put forth an analysis workflow which provides this

accessibility for everyone to reproduce any kind of analysis. However, they fail to provide the same level of convenience for the researcher who created an analysis in the first place. Setting up the workflow eats up a considerable amount of the researcher's time, which they may rather spend on advancing research. This additional effort offsets the increase in productivity, promised by reproducibility, which I regard as most significant in the workflows adoption. Persuading researchers, who find the meta-scientific argumentation noble but impractical, do not care about it or even oppose it, requires concrete, practical benefits. Luckily, most of this setup process may become automated, letting the researcher enjoy the workflows advantages while decreasing the efforts necessary to achieve them. Providing a version of the analysis workflow by Peikert & Brandmaier (2019) that is easier to use and more accessible is the goal of this thesis and the herein presented `repro`-package for the R programming language (Peikert et al., 2020).

2 Technical Solutions

This section summarises the workflow proposed by Peikert & Brandmaier (2019; see also The Turing Way Community et al., 2019 for a very similar approach). They argue that to ensure reproducibility, publically sharing code is not enough. Instead, reproducibility has to rest on five pillars:

1. **file management** a folder containing all files, referring to each other using relative paths
2. **literate programming** a central dynamic document, that relates code to thought
3. **version control** a system in place that manages revisions of all files over time
4. **dependency management** a formal description of how files relate to each other
5. **containerization** an exact specification of the computational environment

These pillars stipulate the relations between thought, code and data with their change over time and environment and hence reach all requirements of reproducibility through being:

1. understandable to other researchers,
2. transferable across machines,
3. conserved through time.

While comprehensibility to the scientific community, is probably the most crucial goal, it is the most difficult to achieve. That is because as a non-technical requirement, no set of rules may assure its fulfilment (though clear writing[1] and clean code[2] certainly help). Transfer and conservation, on the other hand, are problems with technical solutions.

Peikert & Brandmaier (2019) propose to use a combination of RMarkdown, Git, Make, and Docker, because they are the most popular tools for users of the R programming language (R Core Team, 2020) and provide the high level overview seen in Figure 1 .

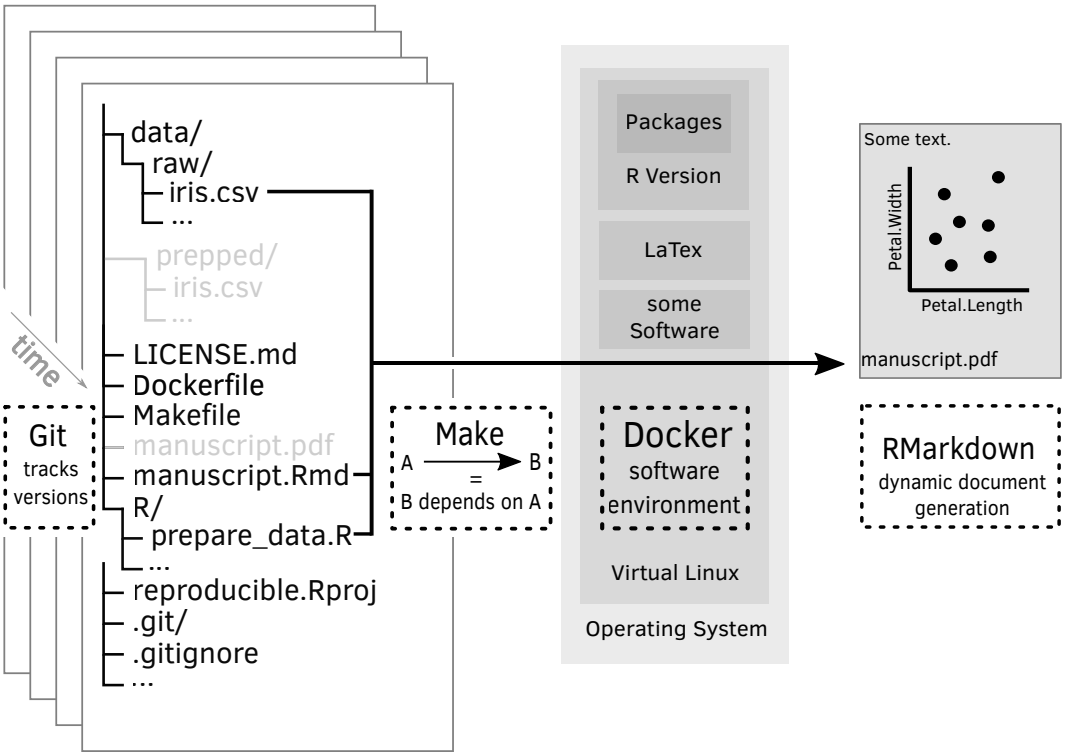

Figure 1: Overview of the interplay of RMarkdown, Git, Make, and Docker

However, they stress that any combination of tools is suitable as long as it facilitates the above pillars.

Each of the following sections first raises a challenge for reproducibility, then outlines a conceptual remedy along with a concrete tool and concludes how they relate to the package `repro`. The relation of these tools with `repro` is then expanded in the next chapter.

[1] Williams (2017) provides some excellent principles for writing clearly.

[2] Martin (2011) proposes a coding paradigm that found widespread use because of its focus on understandability.

2.1 File Organisation

File organization has to meet two challenges. First, the structure needs to be understandable by others, and second, it needs to be self-contained so that it can be moved to another machine.

Adhering to conventions help other people understand how files are organized. For example, the filename `R/reshape.R` follows both standard naming conventions (all lowercase, ends with .R, placed within the R directory) and is meaningful. Contrarily, `myScripts/munge_Data.r` is probably a lot harder to understand and remember for most R-users.

Following two guidelines makes the file structure self-contained:

1. Everything is in one folder.
2. Every path is relative to that folder.

This simple concept of a self-contained folder is facilitated by two R specific tools, RStudio projects and the `here` package (Müller, 2017). The former frees the user from changing the working directory manually; the latter infers absolute paths from relative ones. However, unlike the native R solution, this inference is consistent across operating systems and scripts and RMarkdowns.

The `repro` package comes with a template for an RStudio Project, which sets up a file structure that follows best practices and conventions. This template provides the researcher with a minimal example of an reproducible analysis. The researcher can adept code and files to there need, either by simply changing them by hand or through the modular structure of `repro`.

2.2 Dynamic Document Generation

A clear file structure helps a researchers to better understand how a scientific analysis relates to the code, but a strict segregation of code and document inhibit a full grasp. Providing a direct link, dynamic document generation allows interspersing text with code and its results, producing one human-readable document. The key feature is that every time such a document is rerun, the results are reproduced dynamically. This functionality eliminates errors due to copying and pasting results from statistical software to a text processor. This mistake may be all too common; Nuijten et al. (2016) reports that 50% of papers from the psychological sciences contain an error that may be prevented.

RMarkdown does provide a convenient framework to write such dynamic docu-

ments and render them as a wide range of output formats[3]. In an RMarkdown, three parts can be distinguished:

- one specifying its output and metadata,
- one containing code, and
- one with descriptive text.

Each part uses its own language, all of them designed with ease of use and readability in mind. The section containing the output format and other metadata alongside is written in YAML (see the example below). This specification is located at the top, separated by three dashes at the beginning and end of the section. (R-)Code executing an analysis can be placed in a distinct chunk or inline within the text. The former has three backticks on their own line signifying beginning and end. The later is quoted in a pair single backticks. Examples of both methods can be found below. Text, which is not fenced by either three dashes or backticks, is interpreted as literal text written in the Markup language "Markdown". Markdown allows annotating text to signify formattings such as bold, italic, links and the inclusion of images. Such markup is designed to be simple so the documents is well readable even as source files.

The following section shows examples of metadata, code and text, specified as above described, forming a minimal example of an RMarkdown (adapted source code from Xie et al. (2019)/CC BY-NC-SA 4.0):

````
---
title: "Hello R Markdown"
author: "Ross Ihaka & Robert Gentleman"
date: "1997-04-23"
output: pdf_document
---
````

````
This is a paragraph in an R Markdown document.

Below is a code chunk:

```{r}
fit = lm(dist ~ speed, data = cars)
b   = coef(fit)
plot(cars)
abline(fit)
```

The slope of the regression is `r b[1]`.
````

[3]The document you are viewing also results from a collection of RMarkdowns availible as website, PDF and E-book

Resulting in this document:

[fixme]

Undeniably, RMarkdown greatly faciliates reproducibility, but it cannot ensure reproduction. To virtually garuntee reproduction, the `repro` package extends the yaml metadata to incorporate Dependency Management and Containarization into the process of dynamic document creation.

2.3 Version Control

Text, code and results of a scientific document are refined in cycles of many revisions to accommodate highest standards. As changes accumulate, different versions do too, posing a problem for reproducibility as it may be challenging to find out which version of code relates to the final product. One may argue that in the typical publication process, the final product is apparent: the published paper, so only one version is relevant. However, reproducibility may be crucial even before publication as part of collaboration and the peer-review process. Also, recent trends in the publication process like preprints, open review, registered reports and post-publication review, blur the lines between published and unpublished.

To organize different versions as changes accumulate across the phases of a project across machines and users is a well-known challenge in software development. This challenge is met by a high degree of automation that keeps track of different versions and has advanced facilities to compare and merge them.

One such version control software is Git. Git tracks versions of the project folder by taking snapshots of a given state called commits. Each commit has a unique id, called a hash, a short description of the changes made, called commit message and a link to the previous commit. This linking creates a "pedigree" of versions where it is easy to see how things have evolved. Going back in time to a specific version only requires to know the hash of the commit. To mark commits as special milestones, they can be tagged, e.g. as preregistration, preprint, submission or publication.

While mastering Git requires some experience, most of the time, only four commands are needed, which may be accessed through RStudio's Git interface:

git add take a snapshot of the given file
git commit create a commit of all added files
git push upload recent commits to a server
git pull download and integrate recent commits from the server

While a few other commands are necessary to set up Git in a given project directory, this work is done by the `repro`-package.

2.4 Dependency Management

In an analysis, the results depend on code which in turn depends on data. However, seldomly the data is analyzed as is, but some code is dedicated to preparing it. Most likely, each analysis needs a slightly different version of the data. An analysis of missingness requires the missings to be retained, but some statistical models do not allow that. Or the modelling software requires data to be differently shaped, then the plotting library. It is also often the case that one analysis is based on the output of another and so forth. As these relations can become quite complicated, it is necessary to make them explicit to avoid confusion. Dependency management provides a formalism that describes how files depend on other files. More specifically, it provides an automated way to create files from other files, e.g. it automatically generates a cleaned version of the data, by relying on a cleaning script and the raw data. Such relations may be layered; hence, if a plot requires this cleaned dataset, first the cleaned dataset and then the plot is generated automatically. Such structure allows saving considerable computing time, as dependencies are not generated again if they already exist, but only if one of their dependencies has changed. In this example, upon recreation of the plot, the cleaned dataset is not generated as long as the cleaning script and the raw data remain unchanged. Such intelligent behaviour is most useful when the preprocessing requires a lot of computing time as is typical in neuroimaging or machine learning.

Make is a tool for dependency management, while originally designed for the compilation of programs, it is now increasingly recognized as a tool for reproducibility. It allows for all features above and more as it is an own programming language.

However, the repro package provides a much-simplified interface to the essential features, eschewing the need to learn yet another language.

2.5 Containarization

Most computer code is not self-contained but needs libraries and other software to work (e.g. the R programming language or packages). These external dependencies pose a risk for reproducibility because it may not be clear what besides the code and data is necessary and how to install it. Even when all needed software and their exact versions are recorded meticulously, it may be a challenge to install

them. First, it is difficult to maintain different software versions on the same computer and second it may be unclear how to obtain an exact copy of some years old software version. Setting up a computer exactly as someone else is difficult enough, but replicating some other computer how it was years ago is at best painstaking.

To overcome this challenge, the software environment of a project needs separation from the rest of the software environment. Technically such separation is called virtualization because one software environment is hosted on another. Such virtual environment allows each project to have its own software environment without interfering with each other. Hence, such setup is ideal for conservation and can be easily recreated on another machine.

Docker allows virtualization of the whole software stack down to the operating system, but in a much more lightweight way than traditional virtual machines. This lightweight but comprehensive virtualization is called containerization. Containers save storage by being based on each other enabling reuse. Hence if two containers are based, e.g. on a container for the same R version, they only use the storage they need for the different R packages. Containers are created from a plain specification called `Dockerfile`, that defines on which container it should be based and what software should be installed within it.

The `repro`-package automatically infers which packages are needed and creates an appropriate Dockerfiles and the container from it.

3 Workflow

The `repro` package is designed to streamline the researchers workflow. It helps researchers to setup, create, reproduce and change an analysis with little more than a simple mental model. To that end it stands on the shoulders of giants and is providing only a minimal layer of abstraction around the beforementioned tools. While there is a huge variety on how researchers approach an empierical study and its analysis, figure 2 provides an idealized workflow.

The structure of the chapter mirrors this workflow and explains step by step how to:

1. setup the required software
2. reproduce an analysis that follows the here proposed standards
3. apply and publish changes
4. create a reproducible workflow from scratch

`repro` supports several alternative software implementations for each step and

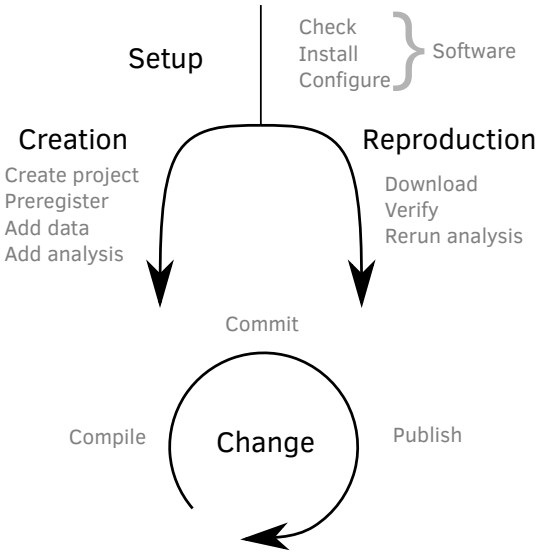

Figure 2: Schematic illustration of a reproducible workflow.

assembles them into one coherent workflow. This modular structure is inspired by the usethis-package (Wickham & Bryan, 2020). The here described steps follow the recomandatations of Peikert & Brandmaier (2019), and hence combine RMarkdown, Git & GitHub, Make and Docker.

3.1 Setup

Reproduction, change, and creations of an analysis require the user to have software installed that is specific to the workflow they choose, but independent of the analysis. A set of functions (following the pattern `check_*`, for a complete list see `help(check)`) help users to verify that everything they need is installed and correctly configured. If `repro` detects that something is not installed or configured it advises users step by step on how to resolve the issues for their specific software plattform. Currently, it supports all major operating systems (Windows, OS X, Linux).

First users have to install `repro`; the following code snippet installs repro from GitHub

```
# check if remotes is installed, if not install it
if (!requireNamespace("remotes")){
  install.packages("remotes")
}
# install repro
# "package::function" means to use a function
# without loading the whole package
remotes::install_github("aaronpeikert/repro")
```

If repro is installed one may load it via:

```
library("repro")
```

Then users can check if the required software is installed. The workflow by Peikert & Brandmaier (2019) requires Git (and GitHub), Make and Docker. Consequently, the following commands check if the user has setup all the requirements:

```
check_git()
```

```
## v Git is installed, don't worry.
```

```
check_github()
```

```
## v You and GitHub are on good terms, don't worry.
```

```
check_make()
```

```
## v Make is installed, don't worry.
```

```
check_docker()
```

```
## v Docker is installed, don't worry.
```

If everything is setup users can proceed to reproduce an analysis that conforms to this workflow.

If not, e.g. because Docker is not installed, users get an informative message appropriate for their plattform (here Windows).

```
check_git()
```

```
## x Git is not installed.
## i We recommend Chocolately for Windows users.
## x Chocolately is not installed.
## * To install it, follow directions on:
##   'https://chocolatey.org/docs/installation'
## i Use an administrator terminal to install chocolately.
## * Restart your computer.
## * Run `choco install -y git` in an admin terminal to install Git.
```

3.2 Reproduction

GitHub, Make and Docker are sufficient to reproduce this document. So everything is setup to download the source files of this document, rerun the code within it, and verify its results.

The following command uses Git and GitHub to:

1. create a copy of the project, called fork, in your GitHub account,
2. download this copy to your computer
3. and verify that all files are intact and opens them in a new RStudio instance.

```
usethis::create_from_github("aaronpeikert/repro-thesis",
                            tempdir(),
                            fork = TRUE)
```

If executed, this code opens a new R session and therefore all code from here on needs to run in the *new* session.

It is tempting to automate the reproduction part completely and have a `rerun()` function that figures out what to do and does it for you. However, I decided that the reproduction musst be easily possible without the `repro` package. This decision prevents that long term reproducibility depends on the availibility of the package. While I am certain that Git, Make, Docker will be availible for years to come, I cannot say the same about this package. To compromise the needs of long term support and usability, `repro` offers advice about what to do, but stops short of actually doing it.

What steps one has to take, depends on the tools choosen to implement dependency management. This tool basically determins the "entrypoint" for an analysis. To detect the entrypoint `repro` follows simple heristics, which stem from what R users tend to use. These conventions are amibiguie, but the clearest entrypoint is a `Makefile`. If no `Makefile` is availible the alternatives are either a central dynamic document (RMarkdown, Jupityer Notebook) or a main script (R, Python, Oktave, Shell). In these cases one can only guess from filenames like `manuscript.Rmd`, `analysis.Rmd`, `paper.Rmd`, `run.R` or `analysis.R`.

To recreate this document you have to follow these steps:

```
# because this is a new R project / session, reload repro
library("repro")
rerun(cache = FALSE)
```

The argument `cache = FALSE` ensures that everything that can be recreated is recreated even when nothing was changed.

It is difficult to verify that an analysis was reproduced. One could accept as a minimal standard, that the analysis is rerun error free or as a maximal standard that the resulting documents are exactly the same. Neither solution strikes the right balance, because error free does not imply the same results, while comparing binarie files often leads to spurios differences. Currently, researcher need to revert to manual checking and common sense to verify a succesfull reproduction. An automated verification procedure would require the researcher to explizitly state which results need to be identical. Then a software solution could track changes for only these digital objects and flag mismatches.

3.3 Change

For a researcher, reproducing an analysis and verifying its results, is often only a first step to make intentional changes. How researcher contribute to a project lays strictly outside the realm of reproducibility, but warrents discussion because easy collaboration is one of the biggest practical advantages of reproducibility. That the main beneficiary of this advantage is the researcher collaborating with its past self is a pun in the open science community that bears some truth. However, the workflow of an external researcher contributing is more complex. It is quite a challange to collaborate under circumstances, where people do not know each other. The core challange is to allow the original creator full control over changes without burden them to much. This problem confronted the open software comunity from its very beginning and they came up with the following solution. A contributer first creates a public copy, makes and tracks changes to it and then asks the original owner to incopareate the changes. In the terminology of GitHub, the public copy is a "fork", the tracked changes are "commits" and the call for including the changes is a "pull request".

Working with pull requests is easy, thanks to the `usethis` package. If you reproduced this document, you could make changes to it—which could be something trivial, like correcting spelling—and ask me to incorporate them. You can initialze a pull request with:

```
usethis::pr_init()
```

Then you can change files as you like and track them with Git. You should make sure that the analysis is still reproducible with:

```
rerun()
```

```
## x This function is not implemented yet.
```

```
## NULL
```

If you are satisfied with the changes you made, you can trigger the pull request with:

```
usethis::pr_push()
```

If I also find the changes to my liking, I can incoperate the changes on GitHub. If not, the changes can be discussed in the pull request or I could make ammends before merging them.

Such distributed workflow, allows a much more controlled way of collaboration as opposed to mail back and forth. This higher level of control fits well into the high standards of scientific work. However, the more important aspect is, that this kind of collaboration scales well for many collaborators (Git was originally devoloped for the collaboration on the linux kernal, where as of 2017 more then 15.000 developers contributed code (The Linux Foundation, 2017)). Empierical studies require a lot of work, which is usally distrubuted on many shoulders. As the authors carry the responsibility for the overall corectness, they ought to vet every single contribution.

Affirming the correctniss of a contribution can be partly automated by affirming successfull reproduction. Such automatic checks of changes are part of a software developing process, called continues integration. Continues integration runs code in cloud computing environments that asserts the correctniss, when changes are pushed to GitHub. In many ways continues integration is the logical next step for reproducible workflows. Because much effort was allready invested to ensure reproducibility across machines it is easy to move the analysis to a continues integration tool.

Hence if you created a pull request, the continues integration tool GitHub actions, will rebuild this document, affirming reproducibility and let me see the results of your changes.

3.4 Creation

Reproducing an analysis and creating a reproducible analysis are two very different things. The `repro` packages main strength lays in simplifiyng the creation. First the repro package comes with a minimal, but comprehensive template inkluding an example RMarkdown, R-script and data. This template can be accessed from within RStudio © via “File” -> “New Project” -> “New Directory” -> Example repro template or from any R console via:

```
repro_template("path/to/new/project/")
```

Repro infers the dependencies to data and external code as well as required packages from the `yaml` metadata of the RMarkdowns. Because data analytic projects have a certain structure this markup can be much simpler than writing `Dockerfiles` and `Makefiles` by hand. While Docker allows to install abitrary Software, an analysis in R likely only needs R-packages. Similarly, Make allows to run any software, but an analysis in R only needs to execute R-Scripts and render RMarkdowns.

Hence, a simple addition to the metadata, like in the following example, contains everything neccesary to infer a complete `Dockerfile` and `Makefile`:

```
repro:
  packages:
    - usethis
    - fs
    - aaronpeikert/repro@d09def75df
  scripts:
    - R/clean.R
  data:
    mycars: data/mtcars.csv
```

The function `automate()` creates a `Dockerfile` and a `Makefile`, which both comply with all recomendations in Peikert & Brandmaier (2019). Strictly speaking it creates four Dockerfiles and three Makefiles. Most of the files a created in the `.repro` directory and then assembled into the main `Dockerfile`/`Makefile` at the toplevel. One `Dockerfile` contains the base docker image, including the R version and the current date and another `Dockerfile` contains only the R packages. It also creates one where the user can manually ammend software installation or setup steps that are not covered by `repro`. The `Makefiles` are similiarly seperated, with one dedicated to RMardowns and another for the logic that executes the make commands in the container.

The `automate()` function is designed to simplify the workflow proposed by Peikert & Brandmaier (2019) as much as possible. Such simplification mean inevidibly to restrict the freedome of the user. While they can still do everything in the realm of Make and Docker, this approach does not allow other reproducibility software to be used. Users, which need more control, can instead rely on the modular nature of `repro`. Each component can be added to the project by the `use_*` functions. E.g. `use_make()` adds a basic `Makefile` or `use_make_singularity()` adds a Makefile that is Singularity compatible (Singularity is an altenative to Docker

for High Performance Computing). These functions extend the usethis-package (Wickham & Bryan, 2020), which was originally designed to faciliate package develpomend with reproducibility specific tools.

3.5 Summary

To summarise everthing you need to do to create a reproducible project in code:

3.5.1 Install `repro` package

```
if (!requireNamespace("remotes")){
  install.packages("remotes")
}
remotes::install_github("aaronpeikert/repro")
library("repro")
```

3.5.2 Check required reproducibility software

```
check_git()
check_github()
check_make()
check_docker()
```

3.5.3 Configure Project

From template in new folder:

```
repro_template()
automate()
```

Or semi automatic with more flexibility in existing projects:

```
use_docker() # create Dockerfile
use_make_docker() # create docker compatible Makefile
usethis::use_git() # initialize git and add first commit
rmarkdown::draft("pnas_article", # use PNAS template
                 package = "rticles") # requires rticles package
```

References

Andersen, H., & Hepburn, B. (2016). Scientific method. In E. N. Zalta (Ed.), *The stanford encyclopedia of philosophy* (Summer 2016). https://plato.stanford.edu/archives/sum2016/entries/scientific-method/; Metaphysics Research Lab, Stanford University.

Announcement: Reducing Our Irreproducibility. (2013). *Nature*, *496*(7446), 398–398. https://doi.org/10.1038/496398a

Boulton, G., Campbell, P., Collins, B., Elias, P., Hall, W., Laurie, G., O'Neill, O., Rawlins, M., Thornton, J., & Vallance, P. (2012). Science as an open enterprise. *The Royal Society*.

Claerbout, J. F., & Karrenbach, M. (1992). Electronic documents give reproducible research a new meaning. *SEG Technical Program Expanded Abstracts 1992*, 601–604. https://doi.org/10.1190/1.1822162

Deutsche Forschungsgemeinschaft. (2019). *Leitlinien zur Sicherung guter wissenschaftlicher Praxis*. https://www.dfg.de/download/pdf/foerderung/rechtliche_rahmenbedingungen/gute_wissenschaftliche_praxis/kodex_gwp.pdf

Epskamp, S. (2019). Reproducibility and replicability in a fast-paced methodological world. *Advances in Methods and Practices in Psychological Science*, *2*(2), 145–155. https://doi.org/https://doi.org/10.1177/2515245919847421

Gigerenzer, G., Krauss, S., & Vitouch, O. (2004). The Null Ritual: What You Always Wanted to Know About Significance Testing but Were Afraid to Ask. In D. Kaplan, *The SAGE Handbook of Quantitative Methodology for the Social Sciences* (pp. 392–409). SAGE Publications, Inc. https://doi.org/10.4135/9781412986311.n21

Gilbert, S. W. (1991). Model building and a definition of science. *Journal of Research in Science Teaching*, *28*(1), 73–79. https://doi.org/10.1002/tea.3660280107

Heilbron, J. L. (Ed.). (2004). The Oxford Companion to the History of Modern Science. *Reference Reviews*, *18*(4), 40–41. https://doi.org/10.1108/09504120410535443

Hutson, M. (2018). Artificial intelligence faces reproducibility crisis. *Science*, *359*(6377), 725–726. https://doi.org/10.1126/science.359.6377.725

Ioannidis, J. P. A. (2005). Why Most Published Research Findings Are False. *PLOS Medicine*, *2*(8), e124. https://doi.org/10.1371/journal.pmed.0020124

Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, *349*(6245), 255–260. https://doi.org/10.1126/science.aaa8415

Kraker, P., Leony, D., Reinhardt, W., Gü, N., & Beham, nter. (2011). The case for an open science in technology enhanced learning. *International Journal of Technology Enhanced Learning*, *3*(6), 643. https://doi.org/10.1504/IJTEL.2011.045454

Martin, R. C. (2011). *The clean coder: A code of conduct for professional programmers / Robert C. Martin* (1. print.). Prentice Hall.

Maxwell, S. E., Lau, M. Y., & Howard, G. S. (2015). Is psychology suffering from a replication crisis? What does "failure to replicate" really mean? *American Psychologist*, *70*(6), 487.

Meehl, P. E. (1990). Appraising and Amending Theories: The Strategy of Lakatosian Defense and Two Principles that Warrant It. *Psychological Inquiry*, *1*(2), 108–141. https://doi.org/10.1207/s15327965pli0102_1

Meehl, P. E. (1978). Theoretical risks and tabular asterisks: Sir Karl, Sir Ronald, and the slow progress of soft psychology. *Journal of Consulting and Clinical Psychology*, *46*(4), 806–834. https://doi.org/10.1037/0022-006X.46.4.806

Müller, K. (2017). *Here: A simpler way to find your files*. https://CRAN.R-project.org/package=here

Nosek, B. A., Ebersole, C. R., DeHaven, A. C., & Mellor, D. T. (2018). The preregistration revolution. *Proceedings of the National Academy of Sciences*, *115*(11), 2600–2606. https://doi.org/10.1073/pnas.1708274114

Nuijten, M. B., Hartgerink, C. H. J., van Assen, M. A. L. M., Epskamp, S., & Wicherts, J. M. (2016). The prevalence of statistical reporting errors in psychology (1985–2013). *Behavior Research Methods*, *48*(4), 1205–1226. https://doi.org/10.3758/s13428-015-0664-2

Open Science Collaboration. (2015). Estimating the reproducibility of psychological science. *Science*, *349*(6251), aac4716–aac4716. https://doi.org/10.1126/science.aac4716

Peikert, A., & Brandmaier, A. M. (2019). *A Reproducible Data Analysis Workflow with R Markdown, Git, Make, and Docker* [Preprint]. PsyArXiv. https://doi.org/10.31234/osf.io/8xzqy

Peikert, A., Brandmaier, A. M., & van Lissa, C. J. (2020). *Repro: Automated setup of reproducible workflows and their dependencies*. https://github.com/aaronpeikert/repro

Popper, K. R. (1962). Some comments on truth and the growth of knowledge. In E. Nagel, P. Suppes, & A. Tarski (Eds.), *Logic, Methodology and Philosophy of Science Proceedings of the 1960 International Congress* (Vol. 155). Stanford University Press.

R Core Team. (2020). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. https://www.R-project.org/

Rodgers, J. L. (2010). The epistemology of mathematical and statistical modeling: A quiet methodological revolution. *American Psychologist*, *65*(1), 1–12. https://doi.org/10.1037/a0018326

The Linux Foundation. (2017, October 25). *2017 Linux Kernel Report Highlights Developers' Roles and Accelerating Pace of Change*. https://www.linuxfoundation.org/blog/2017/10/2017-linux-kernel-report-highlights-developers-roles-accelerating-pace-change/

The Turing Way Community, Arnold, B., Bowler, L., Gibson, S., Herterich, P., Higman, R., Krystalli, A., Morley, A., O'Reilly, M., & Whitaker, K. (2019). *The Turing Way: A Handbook for Reproducible Data Science*. https://doi.org/10.5281/zenodo.3233986

Tichỳ, P. (1976). Verisimilitude redefined. *The British Journal for the Philosophy of Science*, *27*(1), 25–42.

Wickham, H., & Bryan, J. (2020). *Usethis: Automate package and project setup*. https://CRAN.R-project.org/package=usethis

Williams, J. M. (2017). *Style: Lessons in clarity and grace* (Twelfth Edition). Pearson.

Xie, Y., Allaire, J. J., & Grolemund, G. (2019). *R Markdown: The definitive guide*.