Curating Research Assets in Behavioral Sciences: A tutorial on managing research materials with R Studio & the Git version control system

Matti Vuorre[1] & James P. Curley[1]

[1] Department of Psychology, Columbia University

Author Note

Complete departmental affiliations for each author (note the indentation, if you start a new paragraph). Enter author note here.

Correspondence concerning this article should be addressed to Matti Vuorre, 406 Schermerhorn Hall, 1190 Amsterdam Avenue MC 5501, New York, NY 10027. E-mail: mv2521@columbia.edu

Abstract

Enter abstract here (note the indentation, if you start a new paragraph).

Keywords: keywords

Word count: X

Curating Research Assets in Behavioral Sciences: A tutorial on managing research materials with R Studio & the Git version control system

Introduction

The lack of reproducibility is an increasingly recognized problem across scientific disciplines, and calls for changing the scientific workflow to enhance it have been published in a wide range of research areas, including Biology (Markowetz, 2015), Ecology (Ihle, Winney, Krystalli, & Croucher, 2017), Neuroscience (Eglen et al., 2017) and Psychology (Munafò et al., 2017). However, although exhortations to focus on reproducibility in the scientific practice are now commonplace on the pages of leading scientific journals (Baker, 2016), only a small minority of researchers are familiar with the tools and practices that enable implementing reproducibility in the scientific workflow. Therefore, although there now is a broad consensus that efforts to improve reproducibility are important, and even on some of the tools that may allow to do so, materials instructing researchers in using them are lacking. In this tutorial paper, we present a detailed walk-through of the git version control system for behavioral scientists.

Reproducibility

Consider the following (Ihle et al., 2017, p. 2): Have you ever found a mistake in your results without knowing what caused it? Forgot what analyses you have already done and how (and possibly, why)? Lost datasets or information about the cases or variables in a dataset? Struggled in redoing analyses when new data became available? Had difficulty understanding what data to use or how in a project that you inherited from another researcher? Answering any of these questions in the affirmative suggests that your work might benefit from improving reproducibility (Ihle et al., 2017).

But what exactly is reproducibility and why does it matter? Reproducibility can be defined as follows: “A research project is computationally reproducible if a second investigator (including you in the future) can recreate the final reported results of the project, including key quantitative findings, tables, and figures, given only a set of files and written instructions." (Kitzes, Turek, & Deniz, 2017) In the context of experimental Psychology, for example, a project would mean an experiment or a set of experiments investigating a theory or hypothesis, reported results would be a conference presentation or a submitted manuscript. In this field, key quantitative findings are usually probability values from statistical models, such as p-values, or tables of descriptive statistics such as (differences in) means.

Given this broad definition, here is an example of a non-reproducible project: A published journal article advertises a specific relationship between two variables, to some specified degree of uncertainty, but doesn't provide the raw data or code used to analyse it. An example of a reproducible project, on the other hand, provides a well organized package of i) the raw data supporting the claims made in the article, ii) the computer code (or steps of analysis) required to compute the summary and test statistics from the data, and iii) instructions on how to apply ii) to i), if it is not self-evident. Clearly, many projects fall in between, and can be partly reproducible–e.g. provide raw data but no code.

This definition makes clear an important distinction: reproducibility is not the same as replicability. Traditional methods and results sections in journal articles have focused on ensuring replicability (but not reproducibility) by giving detailed instructions on how to repeat the experiment and collect a data set *like* the original one. Viewed in this light, replicability is a broad methodological issue concerning the epistemology of the scientific claims; whereas reproducibility–the topic of this tutorial–concerns the minimal steps required to allow checking for the validity of the computations required to assert the scientific claim in the first place.

Although they are often conlated [TODO] reproducibility and replicability are, in fact, independent of one another. One might think that reproducibility [TODO Peng 2011?] is a requirement of replicability, but an experiment can be replicated even if the materials of the original study are not available for replicability. Likewise, the materials of a study might allow its results to be reproduced, yet the underlying scientific idea might not be replicable.

Therefore, it is important to keep these two concepts distinct; in this manuscript we are interested in reproducibility.

Challenges to reproducibility

Although reproducibility might at first appear easy, it has in large parts failed in practice. For example, one survey of leading Psychology journals found that 34-58% of articles published between 1985 and 2013 had at least one inconsistently reported *p*-value (Nuijten, Hartgerink, van Assen, Epskamp, & Wicherts, 2015). Upon recalculation, the authors couldn't obtain the same *p*-values from the reported test statistics, and therefore these 34-58% of articles were, to some degree, non-reproducible.

Why, then, do so many research products fall short of the basic scientific standard of reproducibility? We suggest one reason is the poor level of organization and curation of all the research assets (data and code, usually) that play a part in creating the research product. Researchers usually have no formal methods or accepted gold standards for curating their research assets, and collaborative workflows, especially, are difficult to manage in a reproducible manner. When the complexity and amount of materials related to a project increases, it might be difficult to link analysis files to the correct data files, and track the evolution of both code and data throughout the research cycle.

In fact, the effort to organize and curate materials has been cited as an important reason for why data is so rarely shared in Psychology. In one study, researchers asked authors of 141 Psychology articles (with a total of 249 studies) to share their data for reanalysis (Wicherts, Borsboom, Kats, & Molenaar, 2006). 73% of the authors refused to share their data, and Wicherts et al. suggested that they did so because it takes considerable effort to clean, document and organize datasets. How could we make organizing and curating materials and data easier, so as to improve the reproducibility of our science?

Aims of the article

Fortunately for the empirical sciences, challenges related to organizing and curating materials across time, space, and personnel have been solved to a high standard in computer science with Version Control Systems (VCS). In the remainder of this article, we introduce a popular VCS called Git, and illustrate its use in the scientific workflow with a hypothetical example project. In the tutorial below, we provide show how to use the Git VCS to curate your research materials by using text commands from the computers command line, and with a Graphical User Interface called R Studio.

The Git version control system

VCSs are computer programs designed for "tracking changes in computer files and coordinating work on those files among multiple people" (Wikipedia). In this tutorial, we focus on the most widely used (within science) VCS, called Git.

It is important to understand that unlike an operating system's (OS) default file viewer, such as Finder (Apple computers) or File Explorer (?? Windows), Git is not a standalone program for navigating files and folders on a computer. Instead, it adds functionality to the OSs existing file system, by making available a specific set of functions–either executed from the command line, or through a graphical user interface (GUI) / integrated development environment (IDE)–that allow taking snapshots of the project and its files, and distributing the work across multiple computers and users. To begin using Git, users must first download and install the Git software on their computers. Git is free and open source, works on Windows, Macintosh, and Linux OSs (among others), and is used by major software developers such as Microsoft, Google, and Facebook. It can be freely downloaded at https://git-scm.com/.

Installing Git

Even if you already have Git installed (some computers do) it is a good idea to install the latest version[1], which can be downloaded as a standalone program from https://git-scm.com/download. Here, we provide more detailed instructions on installing Git for OS X and Windows, but Linux (and other OS) users can find instructions at the Git website (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git). After installation instructions, we detail how to configure the Git program so that it is ready to be used.

OS X. Many OS X computers already have Git installed, especially those operating OS X 10.9 or higher, but it is good practice to install the latest version. The easiest way to update Git to the latest version is to download the installer from http://git-scm.com/download/mac, and install it like any other program. Once Git is installed, it can be used from the command line (the OS X command line is available through the Terminal app, which is included by default with the OS X install), or through various GUIs. Although it might at first appear intimidating because of the archaic-looking interface, we encourage using Git from the command line.

To operate properly, Git needs to be able to identify its user. You can set these by entering a few basic commands in the Terminal. The following commands are intended to be entered to Terminal or a similar command line interface. To verify the current user information, type `git config --global user.name`, and hit return. This should not return anything, unless a previous user of the computer has set the global Git user name.

To ensure that Git knows who you are, type `git config --global user.name "User Name"` (where User Name is your name), and hit return. If you now re-run the first command, Terminal will return the name you entered as `"User Name"`. The second piece of information is your email, which can be entered by `git config --global user.email "email@address.com"` (where email@address.com is your email address). You can verify that the correct email address was saved by typing `git config --global user.email` in

[1]As of the writing of this article, the current version of Git is 2.13.1.

the Terminal, and hit return. Once this information is entered, Git will know who you are, and is thus able to track who is doing what within a project, which is especially helpful when you are collaborating with other people, or when you are working on multiple computers.

Windows. Windows users can download the Git software installer from http://git-scm.com/download/win. [TODO]

Using Git

The first operating principle of Git is that your work is organized into independent projects, which Git calls *repositories.* At its core, a repository is a folder on your computer, which is version controlled by Git (you can tell if a folder is "monitored" by git by checking if the folder, or any of its parent folders, contains a hidden `.git` directory). Everything that happens inside a repository is tracked by Git, but the user has full control of what is tracked (everything, by default) and when. Because the user(s) has full control of what and when is tracked, there is a small set of operations the users need to execute every time they wish to log something in Git.

Briefly, when you work in a Git repository, Git stores the state of each file that it monitors, and when any of these files change, Git indicates that they differ from the previously logged state. If the user then is happy with the current changes, she can **add** the changed files to Git's "staging area". If the user then is certain that the files added to the staging area are in good shape, they can **commit** the changes. These two operations are the backbone of using Git to store the state of the project whenever meaningful changes are made.

Most importantly, each state in Git's log contains the full state of the files at that point in time. Users can always go back to an earlier version by "checking out" a previous state of Git's log. That's why these programs are called Version Control Systems.

To establish the operating principles of Git in practice, we now turn to a practical example using a hypothetical project that, we feel, reflects the components of a typical project in experimental psychology. To most clearly show the use of Git, we begin with an empty project with no components.

Organizing files and folders

Observers have noted that implementing reproducibility into the scientific workflow is less time-consuming if it is planned from the onset of the project, rather than attempting to add reproducibility to the project after it has been completed [TODO]. To that end, it is important to organize the project keeping a few key goals in mind (here, we broadly follow established guidelines such as Project TIER recommendations): The files and folders should have easy to understand names (avoid idiosyncratic naming schemes), and the names should indicate the purposes of the files and folders.

The first step is to create a home folder for the project. This folder should have an immediately recognizable name, and should be placed somewhere on your computer where you can find it. We call the example project `git-example`. All the materials related to this project will be placed in subfolders of the home directory, but for now `git-example` is just an empty folder waiting to be filled with data, code, and documents.

Initializing a Git repository

Next, you need to navigate to the folder and initialize it as a Git repository. Here, the `git-example` project is in the users `Documents` folder, and we can use `cd ..` in the OS X terminal to navigate up in the folder hierarchy, and `cd <folder>` to navigate into `<folder>`. So assuming that the Terminal opens up in `Documents`, we navigate to `git-example` with

```
cd git-example
```

Then, to turn this folder into a Git repository, type in

```
git init
```

and hit Return. This command initializes the folder as a Git repository, and the only change so far has been the addition of a hidden `.git` folder inside `git-example` (and a `.gitattributes` file; but users can ignore these hidden files and folders).

Adding a file to Git

Every project, and therefore every repository, should contain a brief not that explains what the project is about and who to contact about it. This note is usually called a readme, and therefore our first contribution to this project will be a README file. The README file should be a plain text file (i.e. not created with Microsoft Word) that can be read with a simple text editor. This file can now be seen in the `git-example` folder by using the system file viewer. Because we added this file to a Git repository, Git is also aware of it. To see what files have changed since the last status change in the repository (there clearly has been only this one), you can ask for Git's **status**:

```
git status
```

Which in this case would return

```
Matti [~/Documents/git-example]$ git status
On branch master


Initial commit


Untracked files:
  (use "git add <file>..." to include in what will be committed)


    README
```

```
nothing added to commit but untracked files present (use "git add" to track)
Matti [~/Documents/git-example]$
```

The relevant output returned from executing this command is the "Untracked files:" part. There, Git tells the user that there is an untracked file (README) in the repository. To keep track of the status of this file, we **add** it to Git by using the command `git add` followed by either a `.` (for adding all untracked files) or `README` for only tracking the `README` file.

```
git add README
```

We've now added this file to the staging area, and if we are happy with changes to the file's status (it has been created), we can **commit** the file to Git's versioning system. Commits are the most important Git operation: They signify any meaningful change to the repository, and can be later browsed and compared to one another. As such, it is helpful to attach a small message to each commit, describing why that commit was made. Here, we've created a README file, and our commit command would look as follows:

```
git commit -m "Add README file."
```

The quoted text after the `-m` flag is the "commit message". Entering this command to the command line returns a brief description of the commit, such as how many files changed, and how many characters inside those files were inserted and deleted.

Keeping track of changes to a file with Git

Most importantly, the `git-example` project now keeps track of all and any changes to README. To illustrate, we now add some text to README to describe the project (title, what the project is about, who are involved, who to contact), save the file, and then ask for Git's status with `git status` on the command line:

```
Matti [~/Documents/git-example]$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README

no changes added to commit (use "git add" and/or "git commit -a")
```

Git can now tell that the README file has changed, and we can repeat the add and commit steps to permanently record the current state of the project to Git's log:

```
git add .  # We use the '.' shortcut
git commit -m "Populate README with project description."
```

What does Git know?

The real importance of these somewhat abstract steps becomes apparent when we consider the Git **log**. We can call

```
git log
```

To reveal the commit log of this repository. The main things to notice in the output of this command are that each commit is associated with a unique hash code (long alphanumeric string), which we can use to call for further information (see below); an author (this is where the earlier setup is apparent); a date and time; and the short commit message. Until now, the `git-example` log looks like this:

```
Matti [~/Documents/git-example]$ git log
commit 60cbe5c9b4a78e500314f791080381030577a035
Author: Matti Vuorre <mv2521@columbia.edu>
Date:   Tue Jun 13 17:20:27 2017 -0400

    Populate README with project description.

commit 16c475023ecbc99446164187eeaaab10647ac550
Author: Matti Vuorre <mv2521@columbia.edu>
Date:   Tue Jun 13 17:14:14 2017 -0400

    Add README file.
```

To see what exactly changed in the last commit (the log has latest commits at the top), we can call `git show` with the commit's hash (only relevant parts of output shown below):

```
Matti [~/Documents/git-example]$ git show 60cbe5c9b4a78e500314f791080381030577a035
commit 60cbe5c9b4a78e500314f791080381030577a035
Author: Matti Vuorre <mv2521@columbia.edu>
Date:   Tue Jun 13 17:20:27 2017 -0400

    Populate README with project description.

diff --git a/README b/README
--- a/README
+++ b/README
@@ -0,0 +1,8 @@
+# Example Git Project
```

```
+This example project illustrates the use of Git.

+authors:

+Matti Vuorre <mv2521@columbia.edu>

+James Curley

+2017
```

This output can be investigated for a detailed log of all changes created by that commit. From top, it lists the author of the commit, the commit message, and then the commit's "**diff**" (i.e. what differs in the new version vs the old version of the file.) The current diff shows that 1 file received eight additional lines of text (the part wrapped in @ symbols), and then the additions themselves (the lines prepended with +s).

Although we now understand the fundamentals of using Git to track the states of (and therefore changes to) a repository, this contrived and overly simplistic example doesn't allow full appreciation of the benefits of using Git for version control.

Footnotes

References

Baker, M. (2016). 1,500 scientists lift the lid on reproducibility. *Nature News*, *533*(7604), 452. doi:10.1038/533452a

Eglen, S. J., Marwick, B., Halchenko, Y. O., Hanke, M., Sufi, S., Gleeson, P., … Poline, J.-B. (2017). Toward standard practices for sharing computer code and programs in neuroscience. *Nature Neuroscience*, *20*(6), 770–773. doi:10.1038/nn.4550

Ihle, M., Winney, I. S., Krystalli, A., & Croucher, M. (2017). Striving for transparent and credible research: Practical guidelines for behavioral ecologists. *Behavioral Ecology*. doi:10.1093/beheco/arx003

Kitzes, J., Turek, D., & Deniz, F. (2017). *The Practice of Reproducible Research: Case Studies and Lessons from the Data-Intensive Sciences*. University of California Press. Retrieved from https://www.practicereproducibleresearch.org/

Markowetz, F. (2015). Five selfish reasons to work reproducibly. *Genome Biology*, *16*(1), 274. doi:10.1186/s13059-015-0850-7

Munafò, M. R., Nosek, B. A., Bishop, D. V. M., Button, K. S., Chambers, C. D., Sert, N. P. du, … Ioannidis, J. P. A. (2017). A manifesto for reproducible science. *Nature Human Behaviour*, *1*, 0021. doi:10.1038/s41562-016-0021

Nuijten, M. B., Hartgerink, C. H. J., van Assen, M. A. L. M., Epskamp, S., & Wicherts, J. M. (2015). The prevalence of statistical reporting errors in psychology (1985–2013). *Behavior Research Methods*, 1–22. doi:10.3758/s13428-015-0664-2

Wicherts, J. M., Borsboom, D., Kats, J., & Molenaar, D. (2006). The poor availability of psychological research data for reanalysis. *American Psychologist*, *61*(7), 726–728. doi:10.1037/0003-066X.61.7.726