Curating Research Assets in Behavioral Sciences: A Tutorial on the Git Version Control System

Matti Vuorre[1] & James P. Curley[1,2]

[1] Department of Psychology, Columbia University, New York, USA
[2] Department of Psychology, University of Texas at Austin, Texas, USA

Abstract

Recent calls for improving the reproducibility of behavioral sciences have increased attention to the ways in which researchers curate, share and collaborate on their research assets. In this tutorial paper, we explain how version control systems, such as the popular Git program, address these challenges to reproducibility. We then present a tutorial on how to use Git with a graphical interface in the R Studio development environment. This tutorial is especially written for behavioral scientists with no previous experience with version control systems, and covers single-user and collaborative workflows. We also include an appendix with information on advanced Git command line functions. Git is easy to learn, presents an elegant solution to specific challenges to reproducibility, facilitates multi-site collaboration and productivity by allowing multiple collaborators to work on the same source simultaneously, and can be implemented on common behavioral science workflows with little extra effort. Git may also offer a suitable solution to transparent data and material sharing through popular online services, such as GitHub and Open Science Framework.

Keywords: reproducibility; version control; git; research methods; open science

Word count: 7700

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

Introduction

Lack of reproducibility is increasingly recognized as a problem across scientific disciplines, and calls for changing the scientific workflow to enhance reproducibility 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). Some studies suggest that one specific challenge to reproducibility is the ways in which researchers organize, curate, share, and collaborate on the research assets—e.g. data and code used to analyze the data—which support the product's conclusions (Vanpaemel, Vermorgen, Deriemaecker, & Storms, 2015; Wicherts, Borsboom, Kats, & Molenaar, 2006).

Fortunately for the empirical sciences, challenges related to organizing and curating materials across time, space, and collaborators have been solved to a high standard in computer science with Version Control Systems (VCS). In this tutorial article, we introduce a popular VCS called Git, and illustrate its use in the scientific workflow with a hypothetical example project. We show how to use Git with a Graphical User Interface (GUI) implemented in the popular R Studio Integrated Development Environment (IDE; RStudio Team (2016)), and include an appendix for more advanced command line Git functions. We also show how to use Git with the popular online service GitHub for collaborative workflows. Learning how to use Git(Hub) will streamline workflows and help researchers stay better organized, and thereby facilitate reproducibility.

Version Control Systems

Before proceeding with the practical tutorial, it is important to understand what Version Control Systems (VCS) are and what problems they were designed to address. VCS are computer programs designed for tracking changes to computer code, and collaborating on the code with others. They were initially developed by software engineers for writing code collaboratively[1], but are increasingly being adopted to enhance workflows outside computer science. To help understand why, it is helpful to think of "code" more broadly as any text written on a computer: Manuscripts, books, statistical analysis scripts, source code of computerized experiments, and even data files are "code" or have source code, which is just plain text written on a computer, and which can benefit from version control. Going forward, when we use the word "code" in this tutorial, we mean it in this broad sense (e.g. this manuscript's source "code" was written on a computer, and was version controlled.)

For example, computerized behavioral experiments are usually written with computer programming languages such as MATLAB or Python. The experiment's source code (text written by humans but interpreted by computers to e.g. display stimuli to participants) may have multiple authors and go through multiple versions. Keeping track of the versions of the program and changes to the code, and allowing many authors to contribute to it (without breaking the experimental program) are problems that VCSs were specifically designed for. It might be less obvious that writing a manuscript is quite similar, at least as far as the computer is concerned: Multiple authors write multiple versions of a text document, and sometimes previous versions need to be inspected, and the text needs to be "merged"

[1]The software we present below is used used by major software developers such as Microsoft, Google, and Facebook on code bases with hundreds of contributors.

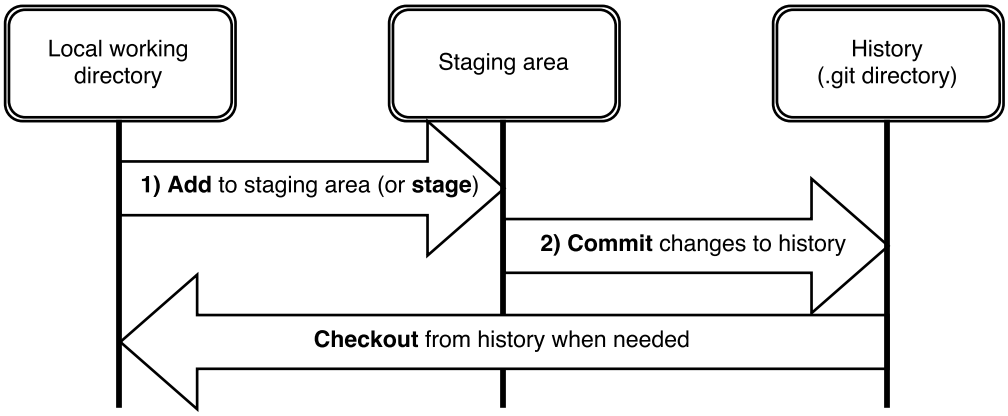

Figure 1. A diagram illustrating the typical Git workflow. Verbs indicated in bold text are Git operations and explained in detail in the main text. In brief, this workflow consists of making changes to files in the project (such as editing a manuscript.) Once the revised version of the file is saved, the user adds the changed file to the staging area (1). Many files can be added to the staging area, if desired. Once the changes in the staging area reflect a conceptual entity, such as edits to an image file and its accompanying caption in the manuscript, the user then commits the changes to Git's history (2). These commits are accompanied with short commit messages that describe the changes made in that commit. Finally, when needed, changes to files can be discarded by checking out an earlier version of the file from Git's history. Figure adapted from https://git-scm.com/book/en/v2/Getting-Started-Git-Basics.

across the many authors. Even less obvious is that datasets are often plain text: In most computerized behavioral experiments, the output data are numbers and text written into a text file. These text files can be version controlled (and researchers would usually not want to see that their raw data files have changed after they were created—VCS allows verifying that they haven't changed because their history is logged.)

The core concepts of version control are that contributors to a project create small checkpoints of the changes they make to the source code—analogous to saving an intermediate version of a file on the computer's hard disk—and then submit those changes to the VCS. The VCS maintains a history of changes to the code between these little checkpoints, and therefore allows coming back to any earlier version by browsing the history. These concepts—i.e. the typical Git workflow—is illustrated in Figure 1.

What Problems Do VCS address?

To understand what VCS is, it is useful to contrast an example scenario with VCS to what we believe is a fairly standard workflow. Consider collaborating with one person on a manuscript that reports results from a data set using some statistical model. In the standard workflow, one person might format the raw data in one way to fit a statistical model, and

then write a draft of the manuscript. This would create three files: `data.csv`[2], `analysis.R`, and `manuscript.doc`. If the coauthor then wished to explore another statistical model, which requires the data in a different format, and then edit the manuscript, she would create three more files: `data_new.csv`, `analysis_new.R`, and `manuscript_new.doc`. This cycle would then repeat as many times as required, each time creating more files, making it more and more difficult for the authors to remember which data was paired with which analysis, and which analyses (and data format) were reported in which version of the paper.

VCS greatly streamlines this workflow: With VCS, there would be only three files (data, analysis, and manuscript), but they would all be under version control, which would keep track of changes to the files and their different versions. Because VCS monitors changes to the files and allows saving a history for each of them, creating new files for every new idea or edit is unnecessary. This lack of duplicity, in turn, possibly reduces room for error in remembering which data file was linked with which analysis, and which manuscript version had the correct numerical results, and so forth. Further, as we explain below, VCS allows many coauthors to work on the files simultaneously, eschewing the need for emailing different versions of the same file to the coauthors, then waiting for them to make changes before continuing with your work. Box 1 shows practical examples of scenarios where VCS (specifically, Git, see below) can be used to improve the scientific workflow. To get a birds-eye view of the Git workflow, before beginning with the details, we suggest readers to inspect Figures 1 and 9 at this point.

Box 1. How Git facilitates the scientific workflow.

How to try different ways of visualizing and modeling data while keeping track of the different versions—and the differences between them?

- Git saves each version of the analysis, allowing testing new features without losing previous versions or proliferating files in the project's directories. All the past versions can be directly compared with each other, and retrieved from the Git's history.

How to work on the same code or manuscript files simultaneously with collaborators?

- Git was designed for distributed collaborative work: Collaborators work on their local copies and "push" and "pull" material to and from each other or a central "repository". Git keeps precise track of who has done what, when, and (possibly) why. Git never loses information or overwrites work with another collaborator's work, but allows for many authors to work on the same ideas simultaneously.

How to share work easily, safely, and in an organized manner?

- Git enforces a common organization scheme among collaborators, making it easier to keep everyone "on the same page" with what goes where, and how to contribute to specific parts of the project. Git projects are shared as a whole, so complex projects with multiple files linking to each other are easy to share. Sharing projects with others is built into Git, and can be facilitated with online services such as GitHub: Once a project uses Git, it can be very easily copied to GitHub, from where others can easily download the entire project onto their local computers.

[2] `.csv` stands for comma separated values, and is a plain text file commonly used to store two-dimensional data.

The Git Version Control System

Version control software has a long history in software engineering, and there are many VCS programs. Some popular ones are Apache Subversion[3], Mercurial[4], and Git[5]. In this tutorial, we focus on Git because it is already increasing in popularity within the scientific community, and is especially good for scientific collaboration because of online tools which allow seamless collaboration even for very large research teams. One of these online tools is GitHub, which we discuss in more detail below. Further, Git is free and open source, works on Windows, Mac, and Linux operating systems (among others).

Git, unlike the operating system's (OS) default file viewer (Mac's Finder, Windows' File Explorer), is not a point-and-click program for navigating files and folders on a computer. Instead, Git adds functionality to the existing file system, by making available a specific set of commands—either executed from a graphical point-and-click user interface (GUI) or from the computer's command line—which allow keeping track of files and their history, and distributing the files across multiple computers and users. Because Git is a standalone program not usually included in standard OS installs, users must first download and install the Git software on their computers.

Installing Git

Even if you already have Git installed (some computers do) it is good practice to install the latest version[6], which can be downloaded as a standalone program from https://git-scm.com/download. For Mac users, the easiest way to install or 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. Similarly, Windows users can download the Git software installer from http://git-scm.com/download/win and install it like any other application. Linux (and other) users can download Git and find further install instructions on the Git website (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git).

After the Git software has been installed, its functions are available to the computer's users through various Git clients. In this tutorial, we show how to use Git with R Studio. We also present more detailed instructions for using Git from the command line in the supplement.

Using Git with a Graphical Interface

The first operating principle of Git is that your work is organized into independent projects, which Git calls *repositories*[7]. A repository is a folder on your computer which is version controlled by Git[8]. Everything that happens inside a repository is tracked by Git,

[3] https://subversion.apache.org/

[4] https://www.mercurial-scm.org/

[5] https://git-scm.com/. The creator of Git, Linus Torvalds (who also is the principal developer of the Linux operating system), named Git after himself as "the stupid content tracker" (McMillan (2005); https://git-scm.com/docs/git.html)

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

[7] For advanced users, Git *submodules* allow linking projects to each other, or organizing more complex projects into projects and their sub-projects (https://git-scm.com/book/en/v2/Git-Tools-Submodules).

[8] There are no visible changes to a folder once it is tracked by Git. Once Git is initialized in a folder, the only change is that a hidden folder, called `.git` is added, but users do not need to interact with it directly.

but you have full control of what is committed to Git's history and when. Because you have full control of what and when is committed to history, there is a small set of operations you need to know[9].

Briefly, when you work in a Git repository (make changes to files within it), Git monitors the state of each file, and when they change Git knows that they differ from the previously logged state. If you are happy with the current changes, you **add** the changed files to Git's "staging area". If you then are certain that the changes in the staging area are desirable, you **commit** the changes. These two operations are the backbone of using Git to store the state of the project whenever meaningful changes are made. Importantly, each commit in the repository's history contains information to recover the full state of the project at that point in time. Users can always go back to an earlier version by **checking out** a previous state from Git's history (Figure 1).

To understand the Git workflow in practice, we now turn to a practical example using a hypothetical project. Git can be added to a project at any stage of the project's life cycle, but to most clearly show its use, we begin with an empty project.

R Studio

R Studio[10] is an integrated development environment (IDE; RStudio Team (2016)) for the R programming language (R Core Team, 2017), and has a graphical module for interacting with Git. An IDE is an interface that bundles together many tools for software development—some users may be familiar with R Studio for conducting statistical analyses. Version control is an essential feature of software development, and R Studio provides a good GUI within its IDE for controlling Git[11].

For this tutorial, we will begin a completely new project using only R Studio, and assume that readers are familiar with the concepts from the above tutorial. Once downloaded and installed, it can be used for accessing the R programming language, and for many operations involved in curating research assets, including a GUI for Git.

Organizing Files and Folders

Implementing reproducibility into the scientific workflow is less time-consuming and effortful if it is planned from the onset of the project, rather than added to the project after all the work has been completed. It is therefore important to organize the project keeping a few key goals in mind (here, we follow guidelines such as the Project TIER recommendations[12]): 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. We illustrate some good and not-so-good practices in Figure 2.

[9] It helps to have a Git command cheat sheet (https://services.github.com/resources/cheatsheets/) printed and taped on your wall, but it contains many more commands than are needed for the basic use of Git in standard Psychology studies.

[10] R Studio is a free and open source IDE, works on Windows, Mac, and Linux operating systems, and can be downloaded from the project's website at https://www.rstudio.com/products/rstudio/download/.

[11] Although in this example we refer to the R programming language, Git can be used through the R Studio GUI even if you do not use R (or any other programming language.)

[12] http://www.projecttier.org/tier-protocol/specifications/

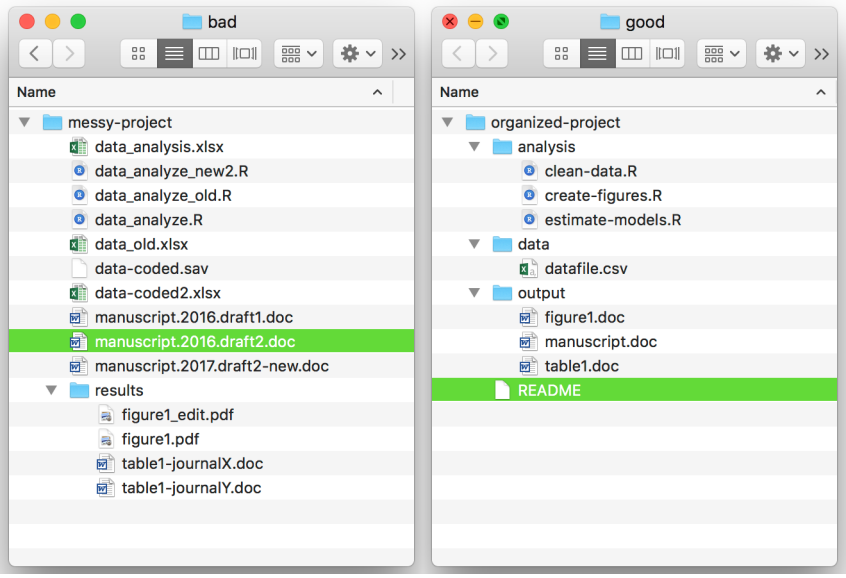

Figure 2. Organizing a project's files and folders helps potential collaborators (including yourself in the future) to quickly and reliably find the correct files. Some bad practices (left panel) include having multiple versions of each file, increasing the potential for e.g. accidentally using the wrong data for the analysis, or forgetting which version was used in the analysis. Better practices (right panel) organize the data to subfolders with meaningful names, and have one file per purpose. These files are then versioned using Git, eschewing the need for multiple files, thereby reducing potential for mistakes.

The project's 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 (and therefore its home folder) `git-example`. Because the folder structure on a computer is easy to think of as a tree, a project's home folder—or any folder that has sub-folders—is also known as the *root* directory. In what follows we use the terms home directory and root (directory) interchangeably.

Managing projects, their folders, and Git repositories with R Studio is centered on the idea of R Projects. To start a new project with R Studio, open the R Studio application, and click File -> New Project. This brings up a dialog (left panel in Figure 3) asking whether to create a project in a new directory, existing directory, or checkout an existing project from a version control repository. Here, we create the project in a new directory. We then choose "New Project" (old R Studio versions may instead have "Empty Project") in the following screen (middle panel in Figure 3). We'll then give a name to the project's home folder (`git-example`) and choose where to save it on the computer. Importantly, we'll also check the "Create a git repository" box (right panel in Figure 3), which will automatically set up a new repository for the project. Clicking "Create project" creates the folder in the specified location, and two files inside the project's main folder.

One of these files is `.gitignore` which we will discuss in more detail below. The other file is an `.Rproj` file, which indicates that the folder is the home folder for an R (Studio) project. Users don't interact with this file directly, but if opened, it is a plain text file

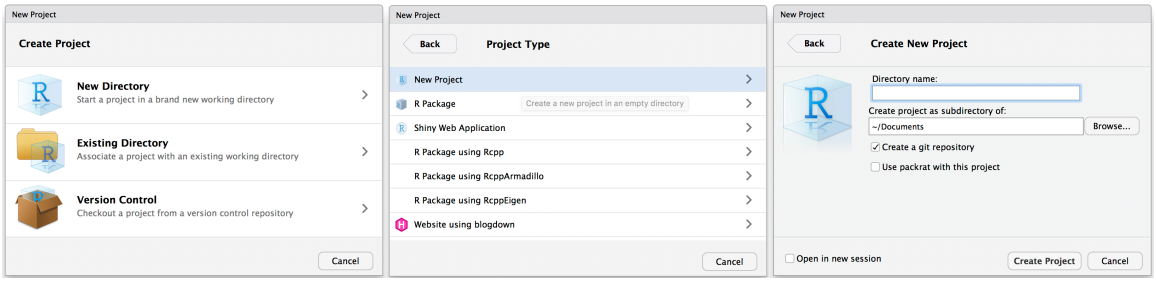

Figure 3. Creating a project in R Studio.

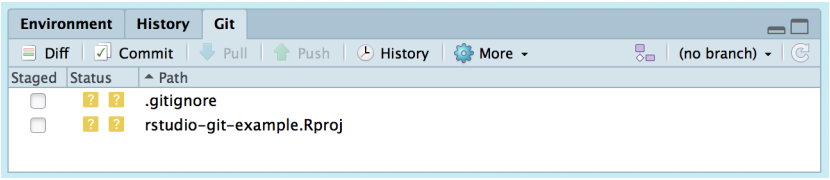

Figure 4. R Studio's Git tab for a newly created project. The Git tab in R Studio has buttons for Git commands covered in this tutorial, such as viewing Git's history, diffs, and commits.

containing the project's settings (these can be modified through Tools -> Project Options in R Studio).

Using Git with R Studio

Once the R Project has been created, R Studio has a "Git" panel in the top-right panel of the GUI (Figure 4). This panel shows the two new files in the repository, and buttons for the main Git commands. Because the project's main folder is initialized as a Git repository, Git monitors any changes within it, and allows you to add and commit these changes (Figure 1.)

Adding Files to Git. To mark this milestone of creating a project, you can commit all the changes so far to Git. To begin committing these changes, click on "Commit" in the Git panel. This brings up another window (Figure 5) where you can select files to add into the staging area. R Studio indicates that there are two new files in the repository with yellow question marks. To **add** these new files to Git's staging area, select the radio buttons in the "Staged" column. This will turn the files' status to a green "A". The text highlighted in green in the bottom part indicates the lines of text that were changed in the selected file (`.gitignore`). These lines are all green because the file is new to Git and therefore each change is an addition (we will discuss the contents of this file in more detail below.)

We have now added the files to Git's staging area, and if we are happy with changes to the files' status, we can **commit** the file to Git's history. Before clicking "Commit", write a short message to the "Commit message" box describing what changed and why. These messages will be important when you later browse the history of your repository. After you click the "Commit" button, R Studio pops up a window summarizing the commit's changes (Figure 6).

Every project (repository) should contain a brief note explaining what the project is about and who to contact. This note is usually called a readme file, and therefore our first

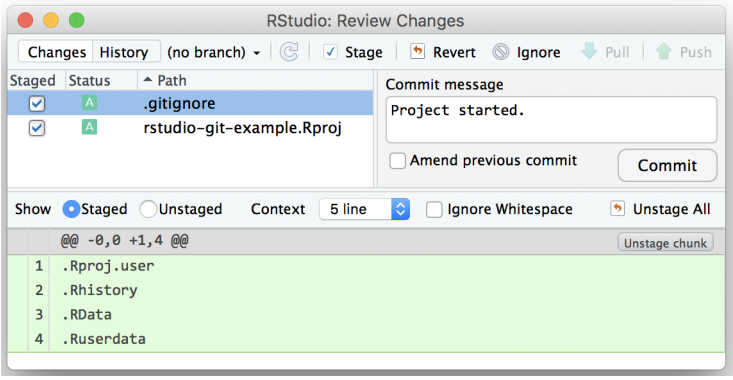

Figure 5. Creating a Git commit with R Studio.

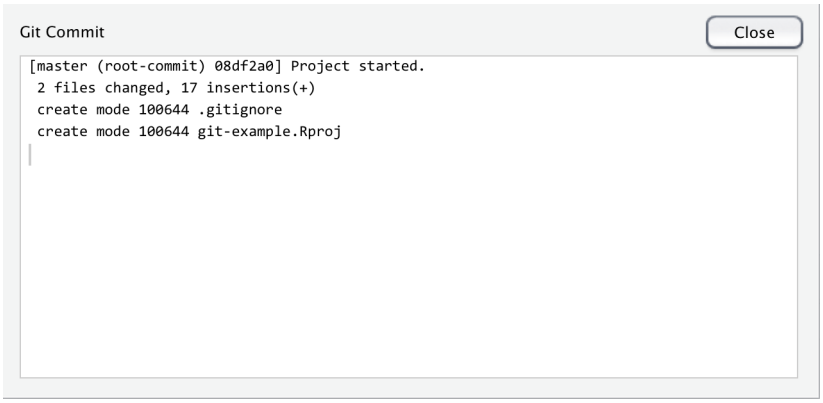

Figure 6. Git's description of changes within a commit. This commit changed two files and inserted a total of 17 lines of text.

proper contribution to this project will be a README file (this file is so important that it has become standard practice to write it in capital letters). The README file should be a plain text `.txt` file (i.e. not created with Microsoft Word or similar) that can be read with a simple text editor[13]. Create this file with R Studio's text editor (click "File" -> "New File" -> "Text File"), and save it to the project's root folder. Once you have saved the file (into the project's home folder), it will be visible in R Studio's Git panel (with yellow question marks, indicating it is a new file). Once you are happy with the README file's contents, stage the file by checking the radio button, write a commit message, and click "Commit". This initial version of the README file is now saved in Git's history, and can be later retrieved. Notice also that after the commit the Git panel in R Studio is empty; there are no changes to the repository.

[13]Plain text has many advantages over proprietary file formats such as Microsoft Word's .docx files. Briefly, plain text is both human and computer readable (.docx are unreadable without a copy of Microsoft Word), is both forward and backward compatible (there will always, and has always been, software capable of reading it), and plain text takes very little space. The file extension (the `.docx` part of a Word file's name, for example) of plain text doesn't matter much, but we recommend using either `.txt` because it is widely recognized, `.md` for markdown syntax, or in the case of a readme file, simply not using any file extension.

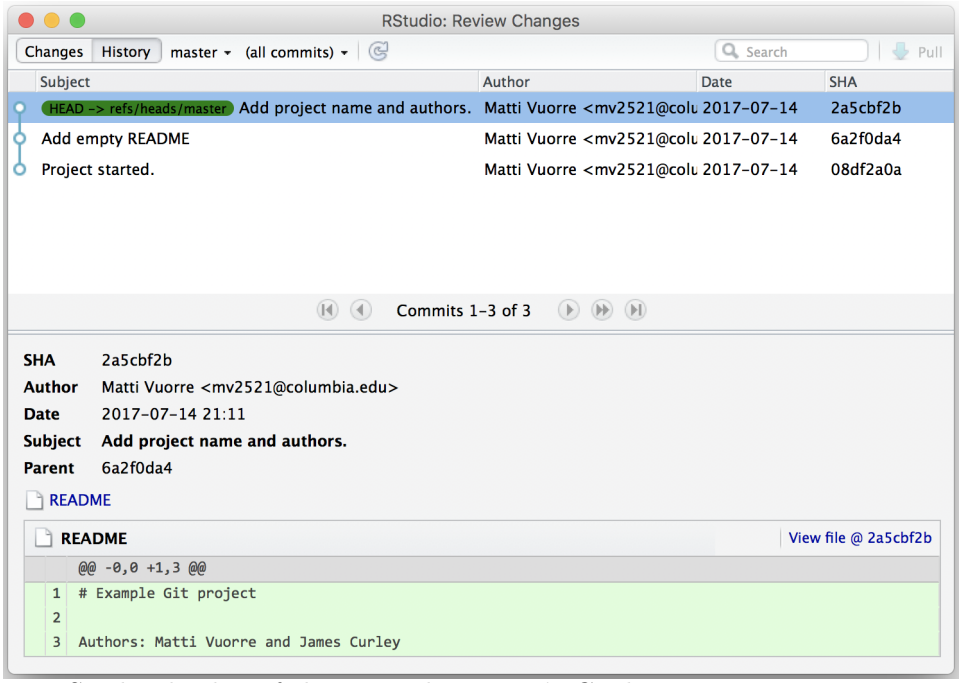

Figure 7. R Studio display of the example project's Git history.

Keeping Track of Changes with Git

The `git-example` project (or rather, Git) now keeps track of all and any changes to README (and the two other files, which we ignore for now). To illustrate, you can change the text in the README file with R Studio's text editor, save the file, and then see that R Studio's Git panel now shows the README file with a blue "M" (modified) flag: Git knows that the README file has been modified since the last commit. It is often useful to know exactly *how* a file has changed, before committing it. To view differences to a file not yet committed, click on "Commit" in the Git panel, and select the appropriate file (README). The bottom panel will then display all the added lines of text in green, and all the removed lines of text in red (see Figure 5). Once you are happy with the changes, you can repeat the add and commit steps from above to permanently record the current state of the project to Git's history.

The real importance of these somewhat abstract steps becomes apparent when we consider Git's **history**. The history contains the exact state of the project at each commit, and allows retrieving previous versions of files. To view Git's history in R Studio, click the "History" button in the Git panel. The following screen's (Figure 7) top panel shows each commit's message, author, date and SHA key, which is a hash code that uniquely identifies each commit. The bottom panel of Figure 7 shows more details about the commit, including the actual changes made to files in that commit. In the current example, the README file received three new lines, shown in green background in Figure 7.

Although you have now seen the fundamentals of using Git to track the states of (and therefore changes to) a repository, this overly simplistic example doesn't allow full appreciation of the benefits of using Git for version control. To better illustrate Git's

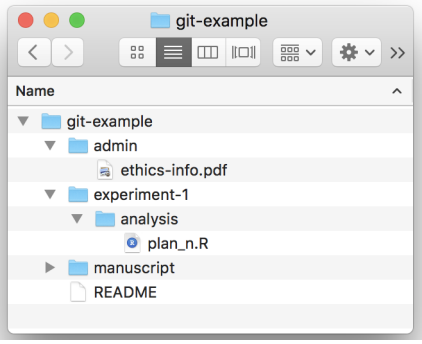

Figure 8. A project with more components.

functioning, we now fast-forward in the hypothetical example project to a stage where more files and materials have been created.

(Slightly More) Advanced Git

After working for a while on the project, you could have added two files to the project. At this point, the project could look like Figure 8. Viewing the Git panel in R Studio now reveals that there are two new files (possible empty folders are ignored): One is a .pdf file with some administrative information (`ethics-info.pdf`), the other one is an R (R Core Team, 2017) script file for a pre-planned power analysis (`plan_n.R`). You would probably like to track any changes to the power analysis script, but the ethics information file isn't something that you need to keep track of—at least for this example. However, it would be a nuisance to constantly keep ignoring it when committing changes to Git, but fortunately Git has an elegant solution to specifying which files to keep track of. Because by default all files are monitored, Git uses a special file for instructing which files are to be *ignored*.

Make Git ignore files. Git uses a special plain text file called `.gitignore` in the home folder of the repository to control which files are to be ignored. R Studio projects create this file automatically, and you can use a text editor to edit it. Notice that the file is *hidden* (by default, not visible in the OSs file viewer), but can be seen in R Studio's **Files** panel. Each row of text in this file should specify a file or a folder (or a regular expression) that Git should ignore. In the current example you could make Git ignore the `admin/` folder entirely (fifth line in the code listing below), and any file with the .pdf extension inside `manuscript/` (sixth line). The first four lines are automatically written by R Studio when creating the project. The example `.gitignore` file would look like this:

```
.Rproj.user
.Rhistory
.RData
.Ruserdata
admin/
manuscript/*.pdf
```

After saving these change to the `.gitignore` file, R Studio's Git panel shows that the `.gitignore` file has changed, and that there are changes in the `experiment-1/` folder. To

reveal which files have changed in the folder, click on the "Staged" radio button next to the folder, which will then select the only changed file in that folder (the `plan_n.R` file). Because there are now two untracked files, which are not specified to be ignored in `.gitignore` (`.gitignore` and `plan_n.R`), and you usually should aim to maintain a clean commit history for the project, you should create two separate commits: One for the `.gitignore` file, and one for the power analysis file.[14]

After committing the `plan_n.R` file to Git's history, you can at any time come back to this commit with Git's history and see what was inside the newly created power analysis file. For instance, if new information suggests that you should change the assumed effect size in the power analysis, you can simply edit and save the file, then add and commit the changes to Git with a helpful message that logs this important event in Git's history.

This possibility of "rewinding history" is especially useful for files that might go under multiple revisions (manuscripts, analysis files), or if you are interested in when and in what order the files were created—and who created or changed them. For example, one might consider committing a power analysis file to Git as a small personal pre-registration of a part of the research plan. To see earlier versions of files, click "History" in the Git panel, and select the commit that contains the version of the file that you would like to see (top panel in Figure 7). Then, find the file in the lower panel of the History window, and click "View file @ SHA" (where "SHA" is the commit's SHA code). This reveals the file exactly as it was at that point in history.

Currently, R Studio's Git panel offers limited functionality in "rewinding history", and to our knowledge the best tools for accessing old versions of the project are Git's command line functions. In the supplemental file we discuss in detail how to retrieve old versions of files, and even old versions of the entire project, using command line functions.

Collaborating

The true advantages of using Git become apparent when we consider projects with more than one contributor. For example, consider a project where data is collected at multiple sites, and the data files are then saved onto a central server, or shared through a service that automatically merges files from multiple sources (such as the popular Dropbox service). If two or more sites accidentally save a data file with the same name to the server (or Dropbox), and these changes are then automatically merged, the later file will simply overwrite the earlier file. Disaster! Alternatively, consider a data analysis where two or more people work simultaneously on some complicated analysis script, and share their work using a similar system as in the above example. If user A and B are making changes to the same file and user B saves the file, user A's version of the file will be overwritten. Disaster!

Git and other VCSs, on the other hand, were specifically designed to allow (and facilitate) multi-site collaboration on arbitrarily complex projects: Microsoft Windows is developed on a Git platform collaboratively by about 4,000 engineers (Harry, 2017). We believe that Git can be especially helpful in scientific collaboration, a topic to which we turn next.

[14] It is entirely up to the user to decide what to commit and when. However, it is best practice to commit often while making incremental changes. Each commit should aim to solve one problem, introduce one new idea, or—more generally—do one thing. This way, when the commit history is reviewed later, it is easy to find and come back to a specific change.

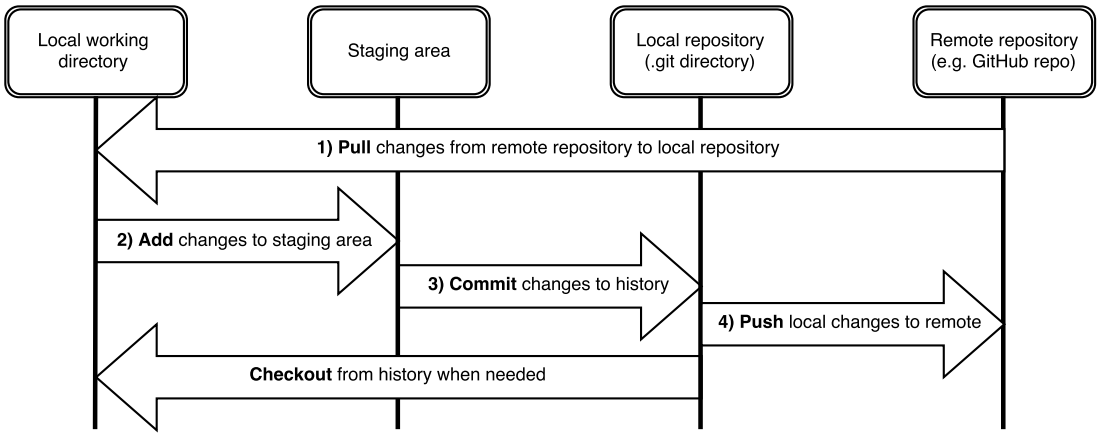

Figure 9. A diagram illustrating the typical collaborative Git workflow with a remote repository (e.g. GitHub). Verbs indicated in bold text are Git operations and explained in detail in the main text.

Overview of the Centralized Git Workflow

There are many ways in which a team could collaborate on a Git project; here we focus on a common one, called "centralized workflow", where one virtual copy of the project is considered central, and all contributors interact with this central copy, instead of directly with each other. In this workflow, a central project hosted on a research team's server, or an online service like GitHub, is used to send and receive information from and to local projects.

Because the word "project" can now refer to virtual copies of projects, they are instead commonly referred to as **repositories**. First, one user creates this central repository, then other users **clone** local copies of it. Contributors, including the one who created the central repository, then work on their local projects as detailed above, and after committing, **push** their changes to the central repository. To get changes which other users have pushed to the central repository, contributors **pull** changes from it. Setting up a centralized Git repository on a research team's private server is relatively straightforward, but because the details vary from team to team, here we illustrate the centralized workflow using GitHub.

GitHub

GitHub is one of the 100 most popular website worldwide, and hosts over 60 million software projects with a total of over 20 million users.[15] We chose GitHub for collaboration because it is already the de facto standard among scientists who use VCS, it offers free private repositories (see below), and repositories from GitHub can easily be connected to projects hosted on the Open Science Framework (https://osf.io).

[15] https://github.com/about.

New GitHub users must first create a free user account at https://github.com. Then, to use GitHub with local Git repositories, you must configure your local Git program to know your name, and especially your email address (GitHub will use your email address for authorization purposes). To configure your local Git program, please refer to the supplemental file, where we show how to use the command line for configuring Git. You must create the GitHub account with the same email address that you used when configuring your local Git (or re-configure your local Git to use your the email address that you used to register for GitHub).

Create a new GitHub repository. To collaborate on a Git project with other, we need to link a local Git repository with a **remote** repository, such as one hosted on GitHub. To do this, we must first create a repository on GitHub.

After you have created an account on GitHub, navigate to https://www.github.com and log into your account. Then click the green "New repository" button. You will first want to give a name to your GitHub repository: The name can be anything, but for consistency we call the GitHub repository `git-example`—the same name as the local project folder. You can then write a short description about the repository in the row below. You can then choose whether the repository should be public or private; we discuss this choice in more detail below, but for now choose "Public". Importantly, because you already have a local repository with a `.gitignore` file and a README file, do not check the "Initialize this repository with a README" (you already created one in the local repository if you followed the steps above on your own computer), and do not add a `.gitignore` or a license. After entering the name, click "Create repository". The next step is to link the new remote on GitHub to the local `git-example` repository.

Connecting the GitHub remote repository to a local repository. Currently, you can only link a new local Git repository to a remote repository in R Studio when you are creating the project (right panel in Figure 3). However, you can connect an existing local repository to a new GitHub remote with two short lines of code in the computer's command line—we show how in the supplemental materials. If you would rather not use the command line, you must therefore create a new project in R Studio, and connect it to a GitHub remote while creating it. Therefore, here we show how to create a new project with R Studio such that it connects to a GitHub remote repository.

Instead of choosing "New Directory" in Figure 3, you create a new local project from an existing GitHub remote (Figure 10). After you have created the new remote repository on GitHub, you need the remote `.git` repository's URL, which you need to use when setting up the new project in R Studio. This is visible on GitHub on the page that appears after creating a new repository. Copy-paste the repository's URL from the address box on GitHub (it will have a `.git` at the end of the URL, e.g. https://github.com/username/reponame.git). Then, in R Studio, click "File" -> "New Project", and select "Version Control" (left panel of Figure 10). This option allows you to create a new local repository by checking out a GitHub remote repository. Click Git (Figure 10, middle panel), then copy-paste the GitHub URL to the "Repository URL" box, and choose an appropriate location for the project ("Documents/" in Figure 10, right panel).

Because you cannot associate an existing local Git repository with GitHub by using R Studio, you've now associated the empty remote (GitHub) repository with a local folder on your computer. Usually, you would do this as soon as a project is started, but because

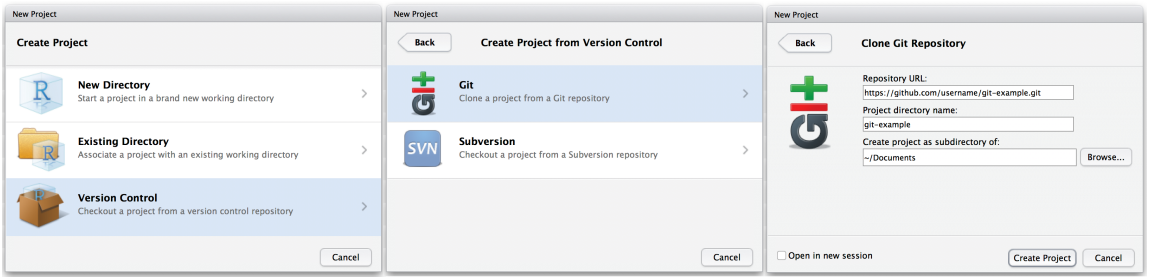

Figure 10. Creating a new local repository by cloning a GitHub repository.

we've already created files and folders in the current tutorial project, we'll copy-paste the materials from the old folder to the new one, which is connected to GitHub. After you have copy-pasted the materials to the new local folder make sure that the contents of the old `.gitignore` file are also copied to the new file. Otherwise you will be committing files into Git's history that you would rather ignore. You can then commit all the files to the local Git repository using the steps detailed above. After adding and committing some files locally, you will have two new buttons in R Studio's Git panel: The local repository can now send (**push**) and receive (**pull**) changes from the remote repository. Click "Push", and then navigate to the repository's GitHub page and refresh the page: You will see all the committed files and folders on GitHub.

Contributing to a central (GitHub) repository. Once another team member has set up Git on their own computer, and signed up for GitHub, they need to **clone** the remote GitHub repository onto their local computer, just as you did in the above steps. Other users can find out the repository's URL by navigating their web browsers to the repository's GitHub address (e.g. https://github.com/mvuorre/reproguide-curate for this tutorial's repository) and clicking the big green "Clone or download" button; the complete address is in the text box. New contributors can then work on their local copies as detailed in earlier parts of this tutorial; making changes, adding, committing, and pushing. After committing their changes, they can update the status of the central Git repo by pushing their changes to it. To push changes, they simply press the "Push" button in R Studio's Git panel.

Note that you can also clone GitHub repositories if you are not aiming to contribute to a project. Hosting projects on GitHub is attractive because other people can easily clone (download) your work to build on, and contribute to, your work.

Obtaining other's changes from the central repo. Just as you must manually push your own local changes to the remote repository, you must also obtain others' changes by **pulling** them from the central repo. Pulling is indicated as the first step in the collaborative workflow in Figure 9, because it is important that you start working on the most up to date version of the project (e.g. you don't want to reinvent the wheel or make unnecessary conflicting changes). Before starting to work on your proposed changes, pull the remote changes by pressing the "Pull" button in R Studio's Git panel.

The way in which users and their local repositories interact with the central repository by pushing and pulling is the cornerstone of collaboration on GitHub, and thoughtful use of these operations allows for complex workflows without any important code (data, ideas in manuscript, analysis code) ever being overwritten. For instance, prior to restarting work

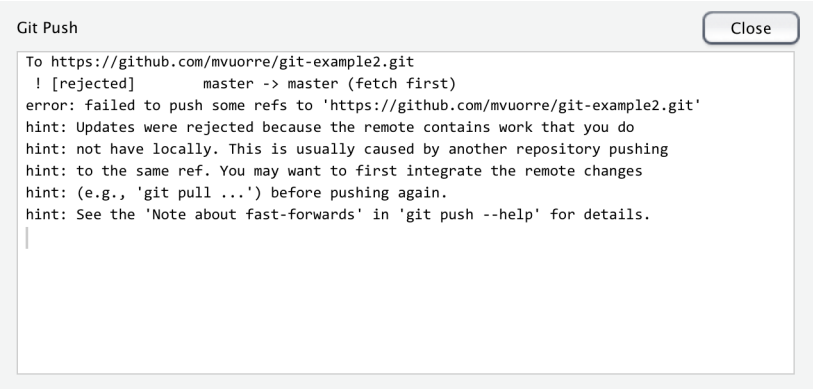

Figure 11. R Studio displays a Git warning if a user attempts to push conflicting changes to the remote repository.

on a project it is often worth checking if any files have changed in the central repository by looking at the project's GitHub page. Most recent commits are listed on top of the file panel, and if someone else has pushed a commit to the GitHub repo, you should pull the changes before starting your work. However, there is no automatic way for a computer to tell what changes to prioritize: If two or more users have worked on the same code and then attempt to push their changes, it is possible that they have made changes which conflict with each other. If this happens, there is no need to worry; you simply need to know how to resolve the conflict.

Resolving conflicts in collaborative work

Many different types of conflicts may appear in collaborative work, such as multiple users creating files with the same name but different content, or multiple users working on the same code (recall that we use the word code to mean any text written on a computer, e.g. text in a manuscript) and creating changes that conflict with each other. We use this latter situation as an example to explain how to resolve conflicts in collaborative work.

Consider the following scenario. Two collaborators, User A and User B, are working on the same project (they collaborate on a repository on GitHub, with local repositories as detailed above.) At some point, they might be working on the same file—such as writing a manuscript together—and find that they have made changes that conflict with each other. More formally, let's assume that both users are making changes to a file, and User B happens to add and commit his changes locally and push them to the central repository before User A does. When User A then attempts to push her changes to the local repository—and the changes are incompatible with User B's changes—a **conflict** will happen. That is just a natural consequence of two individuals working simultaneously on the same idea, and then writing different code in the same location in the file. When this happens, user A needs to first integrate the latest version of the project from the central repository to her local project, such that it reflects both collaborators' edits, and then push the new "merged" version to the central repository. Let's look at what this workflow entails in a little more detail.

First, let's assume that a collaborator (User B) has made changes to the `README` file in the `git-example` project and pushed the changes to the central repository. Meanwhile, if

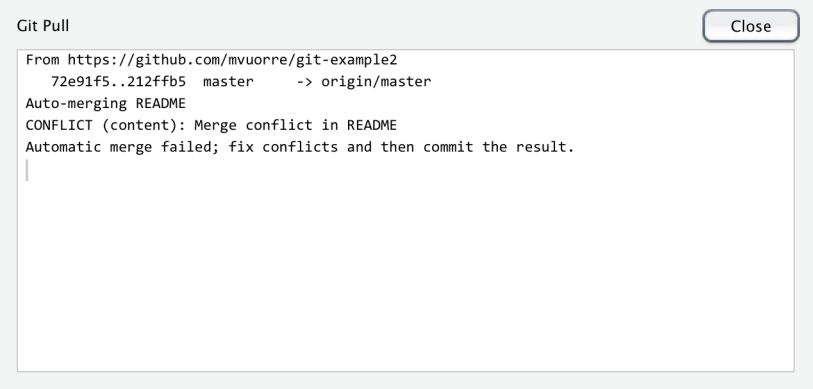

Figure 12. After pulling the most up-to-date remote repository with R Studio, Git attempts to automatically merge the two versions of the file.

User A has committed conflicting changes and then attempts to push them to the remote repository, R Studio will display a Git error message indicating that the push would create a conflict, (Figure 11). In this situation, User A first needs to pull the most current state of the remote repository by clicking "Pull".

After pulling the changes from the remote, Git will automatically merge the two conflicting versions of the file into one file in the local repository. Git will not remove anything, and therefore the user will need to decide which lines of the changed file to save and which ones to discard. Assuming that the two users have made changes to the README file, it can be opened with R Studio's text editor, and might look like this:

```
# Example Git Project
<<<<<<< HEAD
User A's proposed version of text.
=======
User B's proposed version of the line of text.
>>>>>>> 212ffb5de589755ae4fda57fb5af60194283dae8
```

The first line of the file was identical across the two Users' versions of the file, and therefore remains the same. However, after the first line, there is a line (`<<<<<<< HEAD`) indicating that what follows are the to-be-integrated lines of text. Anything after this line up to the `======` are User A's proposed lines of text. After the separating line (`======`) are User B's changes from the remote repository, followed by those changes' SHA key prepended with a `>>>>>>>`. User A can then edit this file however she chooses, using the tags to help her see which are her lines of code (text), and which are User B's. After editing the file, User A must add the changes to Git's staging area (by checking the radio button in R Studio's Git panel), then commit the changes. Once the commit is done, the conflict has been resolved, and user A can push to the GitHub remote.

The options for dealing with conflicts within R Studio's Git panel are somewhat limited, and we present a more detailed tutorial on managing conflicts with Git's command line tools in the supplemental. How these potential conflicts appear depends on how users collaborate with one another, and a detailed explanation of all potential scenarios is outside

the scope of this tutorial[16]. Most importantly, even in the event of conflicts, all committed changes are saved in Git's history and can be retrieved, so experimenting with different approaches to resolving conflicts is safe.

Private or public collaboration?

By default, all GitHub repositories are public: Anyone with an internet connection can use their web browser to inspect the contents of your repository, or even clone it to their computer. This may sound unfamiliar to researchers used to working more privately, and clearly necessitates planning and thought with respect to issues such as data privacy and sharing sensitive materials. However, for many projects—including the writing of this tutorial—we see very few downsides to working "in the open".

There are two alternatives to working in a public GitHub repository: One, which we won't cover here, is to not use GitHub but instead place the central repository on the research team's private server. The second option is to make the repository private on GitHub (this can be done when the repository is first created or afterwards by clicking Settings on the repository's website). Private repositories, and their contents, are only accessible to invited team members, and are therefore ideal for small teams who would like to work without revealing their master plans to the public just yet. For example, you might initially choose to work in a private repository, and only make it public once you feel the material is mature enough for public consumption.

To make a GitHub repository private, navigate to the repository's website with a web browser, and click "Settings", then "Make this repository private". Once one user has set the central GitHub repository to private mode, anyone wishing to clone, push, pull, or view the repo must provide their GitHub username and password. Only if they match an invited team members username and password can the user access the repository. At the time of this writing, GitHub offers five private repositories for free[17].

[16] Covering all different types of file conflicts is outside the scope of this tutorial. Although the instructions provided herein will help in most common use case scenarios, readers can refer to the following websites for more information: https://help.github.com/articles/resolving-a-merge-conflict-using-the-command-line/ and https://www.atlassian.com/git/tutorials/comparing-workflows. You can also resolve conflicts on GitHub (https://help.github.com/articles/resolving-a-merge-conflict-on-github/), and the GitHub customer service is very responsive to users' help requests, which can include questions on code conflicts.

[17] To obtain the free repositories, fill out the request form at https://education.github.com/discount_requests/new. Students with .edu email addresses, can obtain unlimited free repositories at https://education.github.com/pack.

Box 2. Main Git operations

Initialize a local repository.

- Create a local Git repository inside a folder on your computer. In effect, this command turns the local folder into a Git repository.

Clone a remote repository.

- Create a local Git repository from a pre-existing remote Git repository, such as one hosted on GitHub.

Add changes in `file` to Git's staging area.

- Once a file has been edited or created in the local project, it needs to be added to Git's staging area. Once the files have been added, they can be saved to Git's history.

Commit (save) changes from staging area to Git's history.

- A commit creates a snapshot of the project's current state by adding the most recent changes (which have been added to the staging area) to Git's history. Commits should be accompanied with short, descriptive messages to describe the changes introduced in that commit.

Git History.

- Git's history (or "log") is a list of all the commits made in the current repository, and as such is a description of the project's history.

Push (send) locally committed changes to remote repository.

- In the centralized workflow for collaboration as described in this tutorial, collaborators work on their own local repositories, and periodically send the changes that they have made to the central (remote) repository.

Pull (receive) changes from the remote repository.

- To keep the local repository up-to-date with other collaborators' changes, users must periodically pull changes from the remote (central) repository. It is good practice to pull the most up-to-date version of the project every time before making changes, to avoid unnecessary conflicts.

Discussion

We have presented an introductory tutorial on using the Git version Control System for curating and collaborating on research assets in behavioral sciences. The essential Git workflow includes adding and committing incremental changes to a version controlled repository, which can then be collaboratively worked on by many researchers through the online GitHub platform. Box 2 summarizes the main Git operations and their purported use in roughly the order in which they would be used in a typical workflow. In Box 3 we give links to online materials that we have found particularly helpful in learning Git and teaching it to others.

Using Git from R Studio is an especially attractive option for Psychologists because the R Studio IDE offers a complete environment for project management, data analysis, and manuscript preparation with the R Markdown and knitr R packages (Allaire et al., 2016; Xie et al., 2016). Psychologists will be especially interested in the papaja package for creating APA formatted manuscripts (Aust & Barth, 2016). The source code of this manuscript, which was prepared with the papaja package, can be found at https://github.com/mvuorre/reproguide-curate.

Although this tutorial includes enough material to get started, Git (and GitHub) is a vast ecosystem with great opportunities, some of which are further discussed by Perez-Riverol and colleagues (2016). For example, the concept of "Born open data", where research data is automatically posted online upon collection, is made very easy with the Git + GitHub workflow (Rouder, 2016). The challenges to reproducibility are many, and they have only recently received the targeted attention they deserve in the collaborative effort to improve the reliability of empirical sciences. Curating research assets and focusing on the practical aspects of the scientific workflow is important for ensuring the continuity of one's work, and for efforts toward a cumulative and reliable science.

Box 3. Further resources for learning Git

- Basic VCS workflow, an infographic explaining how VCS works (https://www.git-tower.com/blog/workflow-of-version-control).
- GitHub's Git cheat sheet is available in multiple languages and contains the most used Git commands (https://services.github.com/resources/cheatsheets/).
- GitHub's Git glossary describes Git's most common commands (https://help.github.com/articles/github-glossary/).
- TryGit, an interactive website for learning the basics of Git (https://try.github.io).
- Git + GitHub in an R programming context (http://r-pkgs.had.co.nz/git.html).
- Pro Git book, a complete manual of Git (https://git-scm.com/book/en/v2).

Author contributions

MV and JPC designed the format of the tutorial, MV drafted the manuscript, MV and JPC wrote and approved the final version of the manuscript for submission.

Acknowledgments

We thank Travis Riddle and Judy Xu for feedback on earlier drafts of this manuscript. This work was supported, in part, by Institute of Education Science grant (R305A150467). The authors are solely responsible for the content of this article.

References

Allaire, J. J., Cheng, J., Xie, Y., McPherson, J., Chang, W., Allen, J., ... Hyndman, R. (2016). Rmarkdown: Dynamic Documents for R (Version 1.3). Retrieved from https://cran.r-project.org/web/packages/rmarkdown/index.html

Aust, F., & Barth, M. (2016). *Papaja: Create APA manuscripts with RMarkdown*. Retrieved from https://github.com/crsh/papaja

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

Harry, B. (2017). *The largest git repo on the planet*. Retrieved June 20, 2017, from https://blogs.msdn.microsoft.com/bharry/2017/05/24/the-largest-git-repo-on-the-planet/

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

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

McMillan, R. (2005, April 20). After controversy, Torvalds begins work on "Git". *PC World*. Retrieved from https://www.pcworld.idg.com.au/article/129776/after_controversy_torvalds_begins_work_git_/

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

Perez-Riverol, Y., Gatto, L., Wang, R., Sachsenberg, T., Uszkoreit, J., Leprevost, F. da V., ... Vizcaíno, J. A. (2016). Ten Simple Rules for Taking Advantage of Git and GitHub. *PLOS Computational Biology*, *12*(7), e1004947. doi:10.1371/journal.pcbi.1004947

R Core Team. (2017). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/

Rouder, J. N. (2016). The what, why, and how of born-open data. *Behavior Research Methods*, *48*(3), 1062–1069. doi:10.3758/s13428-015-0630-z

RStudio Team. (2016). *RStudio: Integrated Development Environment for R*. Boston, MA: RStudio, Inc. Retrieved from http://www.rstudio.com/

Vanpaemel, W., Vermorgen, M., Deriemaecker, L., & Storms, G. (2015). Are We Wasting a Good Crisis? The Availability of Psychological Research Data after the Storm. *Collabra: Psychology*, *1*(1). doi:10.1525/collabra.13

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

Xie, Y., Vogt, A., Andrew, A., Zvoleff, A., Simon, A., Atkins, A., ... Foster, Z. (2016). Knitr: A General-Purpose Package for Dynamic Report Generation in R (Version 1.15.1). Retrieved from https://cran.r-project.org/web/packages/knitr/index.html