Curating Research Assets in Behavioral Sciences: A tutorial on 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 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.

Version Control Systems

Version Control Systems (VCS; also known as Source Control systems) are computer programs designed for tracking changes to computer code, and sharing the code—and its history—with others. Although initially developed for writing computer code collaboratively, it is easy to recognize the benefits of VCS for scientific production and collaboration as well. For example, behavioral experiments are often written in a programming language, and can include multiple authors and versions. Keeping track of the versions of the program, and allowing many authors to contribute to it (without breaking the code) are problems that VCSs were specifically designed for. By extension, VCS easily adopts to other aspects of the research cycle, such as curating data across computers, laboratories, and experiments; even writing manuscripts is easier when it is recognized that manuscripts (usually) undergo multiple rounds of revision, and that multiple authors might want to keep track of all these different versions.

The core concepts of version control are that contributors to a project create small checkpoints of the changes they make to the source code (or manuscript text, data, etc.), and then submit those changes to the VCS. The VCS maintains a history of all these little checkpoints, and the exact state of the project at each of these checkpoints.

Some popular Version Control Systems are SVN, Mercurial, and Git. In this tutorial, we focus on the most widely used (in science) of these, called Git. Git is especially good for scientific collaboration because of online tools (GitHub) that allow seamless collaboration even for very large research teams.

- Problem: Researchers usually want to try different ways of visualizing and modeling their data, and sometimes it is difficult to keep track of all the different versions.
    - Solution: VCS allows researchers to save each version of the analysis into a history from where it can be easily recalled. This approach also eschews the need for creating multiple copies of the same analysis files with small tweaks, as illustrated here.
- Problem:
    - Solution:

The Git Version Control System

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/. Before we explain the basics of using Git, we present brief installation instructions for Windows and Apple users.

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

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. It is therefore 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 idiosynchratic 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 (to execute code in Terminal, enter the text into the command line and hit Return):

```
cd git-example
```

Then, to turn this folder into a Git repository, execute

```
git init
```

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

(Slightly more) advanced Git

The state of the project now looks like this:

```
git-example/
    admin/
        ethics-info.pdf
    experiment-1/
        analysis/
            plan_n.R
    manuscript/
    README
```

Running `git status` now tells that there are two new files (empty folders are ignored). That is, since the previous commit, two files have been added to the project; one a pdf with some administrative information (`ethics-info.pdf`), the other one is an R script for a pre-planned power analysis (`plan_n.R`). We would certainly like to track any changes to the power analysis script, but the ethics information file isn't really something that we need to keep track of—we can't make changes to it anyway. It would be laborious to constantly keep ignoring it when committing changes to Git (especially if we'd like to commit multiple files simultaneously with `git add .`) Git has an elegant solution to specifying which files to keep track of. Because by default all files are tracked, Git uses a special file for instructing which files are to be *ignored* from tracking.

Make Git Ignore Files. To make Git ignore files, we simply add a plain text file called `.gitignore` to the home folder of the repository. Each row of this file should specify a file or a folder (or a regular expression) that Git should ignore. In our example we want to make Git ignore the `admin/` folder entirely, and any file with the .pdf extension inside `manuscript/. The example .gitignore file looks like this (lines beginning with`#' are comments):

```
# Ignore everything inside the 'admin' folder
admin/

# Ignore all .pdf files in the 'manuscript' folder
manuscript/*.pdf
```

Re-running `git status` now only shows the `plan_n.R` file (and the newly created .gitignore file, which is also under version control, naturally.)

Because there are now two untracked files, and we would like to keep a clean history of what has happened in the project, we create two separate commits: One for the .gitignore file, and one for the power analysis file.

```
git add .gitignore
git commit -m "Added .gitignore file"
git add .
git commit -m "Completed power analysis"
```

After this last commit, if we ever want to know in the future what our planned sample size was, we can come back to this commit with `git log` or `git show` and see what was inside the newly created power analysis file. For instance, if new information suggests that we should change assumed effect size in the power analysis, we 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 the Git memory of the repository.

This possibility of “rewinding history” is especially useful for files that might go under multiple revisions, or if we might be interested when and in what order the files were created. One might consider committing a power analysis as a small personal pre-registration of a part of the research plan.

Collaborating

The true advantages of using a VCS such as 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 these files are then saved onto a centralized 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 (e.g. `data-001.csv`), and these changes are automatically merged, the later file will simply overwrite the earlier file. Disaster!

Or consider a data analysis where two or more people work simultaneously on some complicated analysis. If user A and B have the same file open on multiple computers, after a while when user B saves the file, user A’s version of the file will be overwritten. Disaster!

Git and other VCSs were specifically designed to allow (and facilitate) multi-site collaboration on extremely complex projects, as for example required by leading software companies such as Apple and Microsoft. Next, we’ll illustrate how collaborating on a Git project works in practice.

There are many ways in which a team could collaborate on a Git project (e.g. https://www.atlassian.com/git/tutorials/comparing-workflows), and here we focus on a common one, called the "Centralized Workflow".

Overview of the Centralized Git Workflow

In this workflow, a central repository hosted on a research team's server, or an online service like GitHub (https://github.com/), is used to pull and push (see Table 1) information to and from local repositories. First, one user creates this central repository, then other users clone a local copy (see Table 1). All changes are first created, added, and committed locally as detailed above, and only then **pushed** to the central repository. To get changes that other users have pushed to the central repository, a user **pulls** 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 (https://github.com/about). We chose GitHub for collaboration because it is already the de-facto standard in VCS collaboration among scientists, it offers free private (see below) repositories for students, and has many attractive features, some of which we will detail below.

Anyone wanting to use GitHub must first create a free user account at https://github.com. After the user account has been set up, and the user has logged in, creating a new repository at GitHub is self evident (there is a big green button labeled "New repository").

Create a new GitHub repository. Although we are creating a new repository here, the goal is to create an online "remote" for the running example within this tutorial, and therefore the specific steps are slightly different than what they would be if a brand new repository was needed.

For the current example, after clicking "New repository", we simply enter a name, which can be anything, but for consistency we call the GitHub repository `git-example`. After entering the name, click "Create repository", and the next step is to link the new remote to the local repository. The required steps are detailed on screen when you do this at GitHub, but here we would execute the following command in the command line:

```
git remote add origin https://github.com/<username>/<reponame>.git
```

Where `<username>` and `<reponame>` should be replaced with the user's GitHub username, and the GitHub repository name. Once the connection is set up, we can **push** local changes to the GitHub remote:

```
git push -u origin master
```

The `-u origin master` are only required for the first push, as they set up the connection. For pushing any changes collowing this, simply type `git push` after adding and committing locally.

The team has now created the remote Central repository, and other users can start contributing to it.

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. [TODO actually check that these steps are correct with a "fresh" computer.]

To clone a repository, the new user must first navigate to an appropriate location on the computer where they would like to create the repository on their computer, for example their `username/Documents` folder. Once the appropriate location is found, cloning will create a new folder for the repo inside this folder.

```
git clone https://github.com/<username>/<reponame>.git
```

These new contributors can then work on their local copies as detailed in earlier parts of this tutorial; making changes, adding, and committing. After committing their changes, they can update the status of the Central Git repo by pushing their changes to it:

```
git push
```

Obtaining other's changes from 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. Before starting to work on your proposed change, pull the remote changes with [TODO see if it needs to be rebased]:

```
git pull
```

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 commands allows for complex workflows without any important code (data, ideas in manuscript, analysis code) being ever overwritten. However, there is no automatic way for a computer to tell what changes to prioritize: If two users have worked on the same code, and they both attempt to push their changes, the user to do this later will have to resolve the **conflict**.

Resolving a conflict. If user A has pushed changes to the central repository while user B was working on the same code, user B's push will result in a merge conflict. That is just a natural consequence of two individuals working simultaneously on the same idea. User B simply needs to first merge the changes in the remote repository to her own code by pulling from the remote. Once the conflict has been manually resolved (and user B's code updated with the pull), user B can push changes from her local repository to the central repository. 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. For more information, see e.g. https://www.atlassian.com/git/tutorials/comparing-workflows. The GitHub customer service is also very responsive to users' help requests.

Private or public collaboration?

By default, all GitHub repositories are public, meaning that 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 intimidating to researchers who are used to working in a more private context, 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 shared server. The second option is to change the repository's settings on GitHub (this can be done when the repository is first created or afterwards) to private. 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.

To make a GitHub repository private, navigate to the repository's website with a web browser (e.g. https://github.com/%3Cusername/>), and click "Settings", then "Make this repository private". Once one user has set the central GitHub repository to as private, 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.

Using Git from IDE

Finally, we have above detailed how to use Git from the computer's command line, but not all users are comfortable—although there is no reason not to be!—with this mode of interacting with their computers. We chose to introduce Git from the command line because eventually users will need to use it for more complicated operations (and it is required for setting the user's information); however, there are various graphical user interfaces (GUIs) available for Git, that allow interacting with Git by pointing and clicking buttons with the mouse.

In this section, we introduce how to use the popular R Studio Integrated Development Environment (IDE) for the R programming language for interacting with Git. An IDE is an interface that bundles together many necessary features of 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.

For this example, we will begin a completely new repository using only R Studio, and assume that readers are familiar with the concepts from the above tutorial.

R Studio

[TODO download and install R Studio, cite]

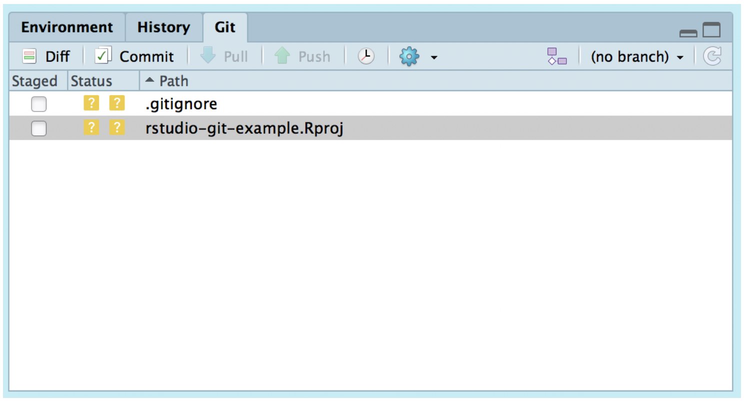
Managing projects (and Git repositories) with R Studio is centered on the idea of R Projects. To start a new project with R Studio, open the R Studio application, and click File -> New Project. This brings up a dialog asking whether to create a project in a new directory, existing directory, or checkout an existing project from a version control repository. Here, we create a new project in a new directory, and choose "New Project" in the following screen. Then we'll give a name to the project's home folder and choose where to save it on the computer. Importantly, we'll also check the "Create a git repository" box, which will automatically set up a new repository for the project (provided you have set Git up as detailed above). Clicking "Create project" creates the folder in the specified location, and two files inside the project's main folder.

[TODO put some screenshots in here]

One of these files is `.gitignore` which we covered above. The other file is and .Rproj file, which indicates that the folder is the home folder for an R (Studio) project. Users don't interact with this file, but when opened, it is simply a plain text file 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" tab in the top-right panel of the GUI. At first, this tab shows the two new files in the repository, and some buttons with familiar looking names ("Commit", "Push", etc.)

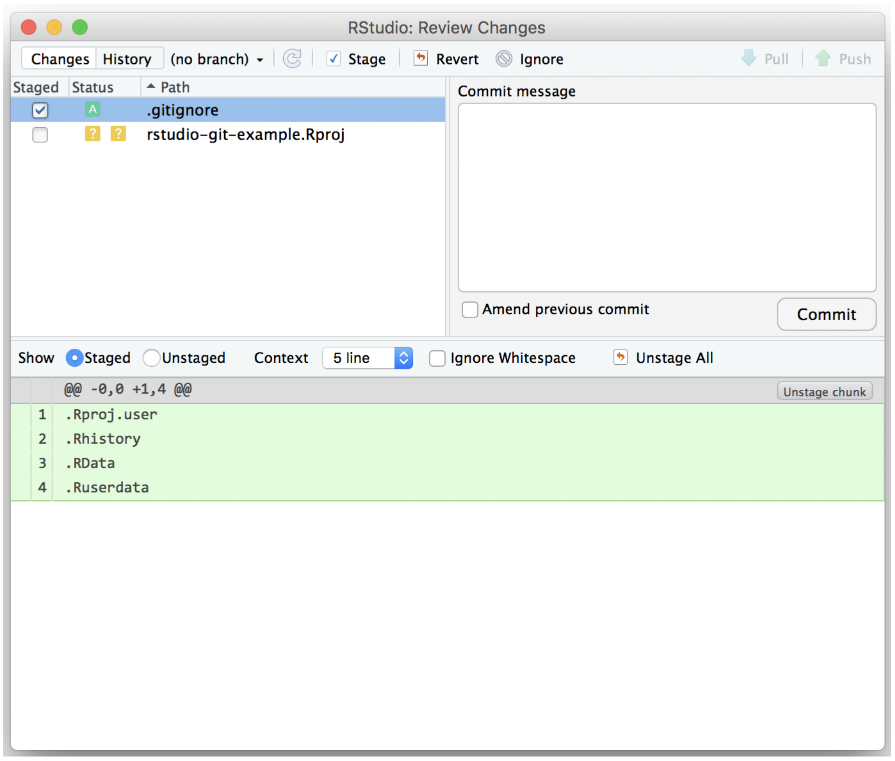


Staged	Status	Path
☐	? ?	.gitignore
☐	? ?	rstudio-git-example.Rproj

Figure 1. R Studio's Git tab for a newly created project.

Add and Commit changes. To mark this milestone of creating a project, let's commit these changes to Git. The workflow is exactly the same as when using Git from the terminal, but we now have some visually appealing helpers. To begin committing these changes, click on "Commit" in the Git tab. This brings up another window where we can select files to add into the staging area (the `.gitignore` file in Figure 2 is staged by checking the box). Once all the desired files (here we chose both) are staged, we write a short commit message as detailed above, and click the "Commit" button.

Add a remote GitHub repository. When creating the R Project, there was an option for creating the project from an existing Git repository. Because we didn't do this, we must now manually instruct Git that this repo should have a remote in GitHub. This is done



Staged	Status	Path
✓	A	.gitignore
	? ?	rstudio-git-example.Rproj

```
@@ -0,0 +1,4 @@
.Rproj.user
.Rhistory
.RData
.Ruserdata
```

Figure 2. Creating a Git commit with R Studio.

exactly as above, and we must use command line functions. After you've created a new repository on GitHub, navigate to the project's folder on your computer with the command line navigator of your choice, and execute the following commands:

```
git remote add origin https://github.com/<username>/<reponame>.git
git push -u origin master
```

After executing these commands, you can use all Git and GitHub features from R Studio without having to use the command line.

Further examples.

Table 1

Main Git commands.

Operation	What it does	Command
Initialize	Turns a local folder into a Git repository	git init
Clone	Create a local copy of an existing remote repository	git clone <url>
Add	Adds files (or changes to file) to Git's staging area	git add <file / .>
Commit	Save changes in staging area to version control	git commit -m '<message>'
Show log	Reveal a timeline of the repository's past commits	git log
Compare	Compare two versions of a file to each other	git diff
Push	Send committed local changes to a remote repository	git push
Pull	Download changes from remote repository to local	git pull

Note. Source: https://help.github.com/articles/github-glossary/ and https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf

Git Summary

Overview of Git Commands

Further resources for learning Git

- Basic VCS workflow, an infographic explaining how VCS works.
- GitHub's Git cheat sheet is available in multiple languages and contains the most used Git commands.
- TryGit, an interactive website for learning the basics of Git.
- Git + GitHub in an R programming context.
- Pro Git book, a complete manual of Git.

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