More with `Git` and `GitHub`

December 11, 2024

Contents

What to (*not*) commit **3**
Untracking files 4

Branching **5**

Differencing & Merging **6**
`diff` for highlighting changes 6
`merge` for collapsing branches 8
Resolving merge conflicts 8
Pull Requests 9

Clones vs. Forks **10**

Going back in time **11**
Revert 11
Reset 11
Recapturing older commits 12
The Nuclear Option 13

Project Management with `GitHub` **13**
Issue tracking 14
Project boards 14

By now you've hopefully gotten into a new routine of working with `Git` (*pull, edit, commit, push*) and are doing well to remember the `Git` motto: *commit early, commit often.* Hopefully your repository is also looking nothing like Fig. 1. Now our goal is to extend our workflow by making it more flexible and adaptable to more diverse project structures and research contexts (such as collaborative projects). To do that, we'll learn the basics of a few more tools that `Git` and `GitHub` offer. Almost all of what we'll talk about below can be done using a `Git` GUI and on `GitHub`, but I might often describe actions using the command-line process for the simplicity and precision that comes with describing commands rather than button icons.

Figure 1: The whole point of a version control system is to avoid your project folder devolving into this! (source: http://maktoons.blogspot.com/2009/06/if-dont-use-version-control-system.html)

What to (*not*) commit

Up to this point, under the guiding principle of reproducibility, we've implied (if not explicitly stated) that everything associated with your project should be kept in your version control respository.[1] That's not entirely incorrect, but it's also not entirely true. For example, `Git` will throw an overridable warning when you commit large files (currently $\geq$ 10mb), and `GitHub` will throw a fatal error when you try to push very large files (currently $\geq$ 100mb)[2]. The latter can happen with specialized data files (e.g., GIS shapefiles). `GitHub` also as a limit on the total size of any one repository (???mb). Sometimes it's therefore necessary to keep some files in a different directory outside of your repository (preferably tracked by `Dropbox` or similar cloud storage).[3] Sometimes it's possible to cut-up your data into smaller pieces. Other times, and more generally speaking, it's possible to reduce the size of a file by saving its contents in a different format: a plain-text format.

What I mean by plain-text format is most easily seen by comparing the contents of a data worksheet saved in a `.csv` file or text saved in a `.txt` file to the same data or text saved in, for example, a Microsoft Excel `.xls` or Word `.doc` file. The former are human-readable when opened and edited in even the most basic of text editors. In contrast, if you were to force open the `.xls` or `.doc` files in the same text editor you'd get a whole bunch of unintelligible and indecipherable computer symbols. That's because the `.xls` and `.doc` files are binary files and have a whole bunch of additional information in them for interpreting, presenting and otherwise doing things (e.g., calculations) with your data and text that requires specialized software.[4] We don't want all that extra stuff for a few reasons:

1. your data should contain nothing but data;

2. we're going to do all calculations by script so as to make them reproducible and readable;

[1] It should go without saying that anything private (e.g., passwords) should never be put in a public repository, but be careful not to do that when we start submitting jobs to High Performance Computing clusters!

[2] If necessary, you could use the Large File Storage extension: https://git-lfs.github.com. Links to additional resources are included in the README file of today's class.

[3] See back to our `Structured Projects` class on using relative links.

[4] `.Rdata` files are also binary files that need R to open them.

3. your text documents should remain readable even in 10 years time and even after Microsoft has yet again updated to save things in a new way;

4. every time you open and save a `.xls` or `.doc` file, a whole bunch of that computer language information is changed even if no content has changed, causing `Git` to save unnecessary copies of the file;

5. we want to make it easy on us to visualize exactly what content has changed in a file since it was last committed (or between any two commits) using a `Diff` tool that we'll talk about below.

So challenge yourself to use plain-text file formats whenever possible. Plain-text files include `.csv` files, `.R` scripts, Markdown `.md` scripts, and LaTeX `.tex` files. Images (e.g., `.jpeg`) and movies (e.g., `.mp4`) are not.[5] Microsoft- and Apple suite files and the like are also (mostly) not plain-text files.

Other types of files you should not commit are temporary files, such as `$.xls` files that are generated only when a program like Microsoft Excel is open, or anything but the primary LaTeX `.tex` (and perhaps the associated `.pdf`) file (because these all get generated each time you compile your document). You should also not commit your `.Rprofile` file (since it contains proprietary API keys). All these temporary file types can be added to your `.gitignore` file.

Untracking files

What to do when you accidentally commit a file that you don't want `Git` to track it (e.g., you forgot to add it to `.gitignore`), or if you change your mind about a particular file and no longer want to track it? If you're using a `Git GUI`, you can probably just right-click the file and select `Untrack` or `Stop tracking`. To remove the file from the staging area via command-line, use `$git rm --cached` *`filename`*. To remove the file entirely (i.e. from the repository), use `$git rm` *`filename`* (without the `--cached`).

[5] Their large file size also makes `Git` and `GitHub` a poor place to store large amounts of image and movie files.

Branching

So far, when we've committed to our `Git` repository, we've been doing so to its *master* (a.k.a *main*) branch. The *master* branch has been our only branch, and every change we've made to a file has been implemented sequentially (i.e. has overwritten what was there before). In other words, our commit workflow has been linear.

But there are often times when you have an idea for doing something differently. You might have an idea for optimizing a section of code (e.g., by vectorizing the use of a function, rather than using a `for` loop[6]), or you might want to try writing a section or paragraph of your manuscript a different way[7]. In both cases you don't want to "break" what's already there and working, but just want to try out an alternative. Branches are the way to do that in `Git`. In the `Git` supporting documents, a branch for trying something out (i.e. adding a new feature) is typically referred to as the "*feature branch.*"

Creating a branch is easy. In your `Git` GUI there's probably a button for doing so at the top of the interface. You can also do it within `RStudio` and on `GitHub`. To create a branch using command-line, type `$git checkout -b` *`branchname`*. `Git` will not duplicate any of your files, but it will keep track of all changes made within the branch (when you commit them to the branch). If you add, edit, or remove files (and commit) while you're on your feature branch, those changes will not appear in your master branch (and vice versa). Thus, if you were to add a file while in your feature branch and were then to switch back to the master branch (using your `Git` GUI, RStudio, or `$git checkout master`[8]), you wouldn't see that new file in your project directory.

The best way to think about your *master* branch is the way a software developer would: the master branch contains your clean, functioning, usable software (or as close as you are to developing it). All other branches are for trying things out. You may end up with many parallel branches for testing out changes to each of several scripts or manuscript sections. Only once

[6]See `Faster Computing` class later.

[7]Or you just got rejected from *journal-that-doesn't-deserve-to-publish-your-work-anyway* and want to reformat your manuscript for another journal, but want to hang on to the original because you may need to move to journal #3 and would want to start from the first journal's version to do so.

[8]Note that the `-b` used previously was only to create the new branch; you don't use it to switch to an existing branch.

you're satisfied with your changes on the feature branch, and have made sure that they work as intended for their specific purpose and in the grand scheme of things (i.e. they don't introduce bugs or affect problems elsewhere in your code), do you bring them back into the master branch and override what was there. That's done by *merging* the feature branch into the master branch (see next section).

The other context in which branches are extremely useful is in collaborative settings where you're working on analyses or a manuscript with others. The workflow is similar to the above in that the master branch contains the not-to-be-broken best of what you've got. Each collaborator creates task-specific branches in which to work (e.g., "I'll work on (re)writing the Methods section," or "I'll work on a script to do this part of the analysis.")[9]. In such collaborative contexts it's often useful to then perform one additional step before merging: to issue a *Pull Request* (in `GitHub`) that asks the other person to review your work before implementing the merge (see below).

Now at some point you will probably end up with more than one feature branch. You might even have created feature branches off other feature branches. Or you might like to go back and look at the branching relationships and history of your repository. While `Git` has a command-line way of visualizing this history, the visualizations provided by `Git GUI`s are more informative and easier to navigate (Fig. 2). You can also use `GitHub` to see the history by going to `Insights>Network` (Fig. 3). (It can take a while for the graph to generate.)

Differencing & Merging

`diff` for highlighting changes

So now you've made changes to your code or manuscript, but before you commit those changes you'd like to compare what you've done to what you had before. That's easy using `RStudio` or any `Git GUI`. In `RStudio` you can see the changes by clicking on `Diff` (or `Commit`) and selecting the specific file from within the staging area. Red lines of code with minus symbols at the start are removals, green lines of code with plus symbols are additions. In `SourceTree` and most other `Git GUI`s, simply selecting the specific file in

[9]Remember to make sure your local master branch is up-to-date before creating a branch by first doing a *Pull* from `GitHub` before you create a branch.

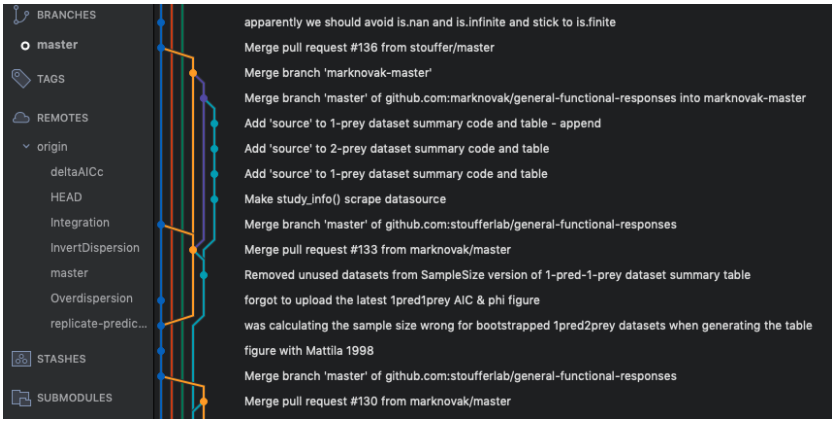

Figure 2: `SourceTree`'s branch visualization.

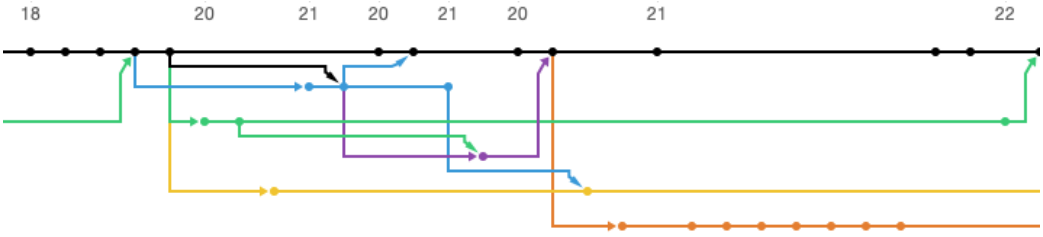

Figure 3: `GitHub`'s "Network" branch visualization.

the "unstaged" area will cause the changed lines to be shown in an adjacent window with similar highlighting.[10]

Similarly, using `Git GUI`s or command-line `Git`, you can compare the changes between your current working files and an older commit, or between any pair of commits within your branch or across branches. In `SourceTree`, simply view the *History* and select the two commits you want to compare using `Ctrl + left click`. For command-line options, see https://git-scm.com/docs/git-diff.[11]

[10] If you use LaTeX, note that `Git` can only differentiate lines ending in a hard-return Hence, any sentences or paragraphs that use line-wrapping won't be differentiated, causing even just a single-word edit to make `Git` think the whole sentence/paragraph has changed.

[11] On a Mac, if you install `XCode`, you'll get an application called `FileMerge` that allows you to compare the line-by-line differences between two files.

`merge` for collapsing branches

You're happy with what's in your feature branch and are ready to bring it into your master branch. In many cases (when there are no conflicts) this is exceptionally easy. In your `Git GUI`, simply switch to the branch (i.e. the master branch) into which you'd like to merge the feature branch, click on the *merge* button, select the feature branch, merge, and commit the new changes with a useful commit message. For command-line, use `$git merge` ***`feature branch`*** and then commit. Remember: If you're working collaboratively, make sure your local copy of the master branch is up-to-date by doing a *Pull* from `GitHub` before attempting to merge (as you did before creating the feature branch). That will help a lot to reduce merge conflicts[12].

Something to consider before you merge branches is your "merging pattern". The most obvious pattern is to merge your feature branch into your master branch. That's the quickest, and introduces little risk when you're working solo or when you're adding new files that are unlikely to entail merge conflicts. The alternative to consider (especially in collaborative settings) is to merge the master branch into your feature branch, test that everything works, then merge the updated feature branch back into the master. This pattern reduces the risk of new bugs in the master that may have be introduced by having two sets of parallel changes. It also reduces the risk of having to resolve merge conflicts on the master branch, which could affect your collaborator's ongoing efforts. The disadvantage of this second merge pattern is that it takes an extra step and results in double the number of merge commits.

When there are no conflicts, `Git` will simply amalgamate[13] the two branches. If you're happy with the result (which you certainly should be if you've followed the second merge pattern), then you can delete the feature branch (e.g., `$git branch -d` ***`branchname`***) or just leave it for posterity.

Resolving merge conflicts

When `Git` notices that the same line of code has been changed on both the feature branch and the master branch subsequent to the feature branch

[12] Just like being clear with your collaborator regarding what sections you're working on so that they don't work on the same sections (though this wouldn't create insurmountable problems).

[13] fanciest sounding synonym for merge I can think of

having been created, it will tell you that there's a conflict. Either you or your collaborator may have made the change on the master; how should `Git` know which of these parallel changes should take precedence? When that happens, you'll have to resolve the conflict manually, then commit those fixes before being able to proceed.

First, identify the file that contains the conflict. Your `Git GUI` should have identified the problem file(s), but you can also use `$git status` via command-line. Once identified, open the file in your text editor and search the file for the conflict marker <<<<<<<. Changes present in the HEAD (in our case master branch) will be after the line <<<<<<< `HEAD`. Next, you'll see =======, which divides these changes from the changes in the feature branch, followed by >>>>>>> *`featurebranch`*. In the example below, one person edited the focal sentence in the master branch to read "feature branch into the master branch", while the other person edited the same sentence in the feature branch to read "master branch into the feature branch".

```
I prefer to merge branches by merging the
<<<<<<< HEAD
feature branch into the master branch.
=======
master branch into the feature branch.
>>>>>>> featurebranch
```

Edit the sentence the way you want it to read (you can change it entirely if you'd like), remove all the lines that have <<<<, >>>> or ==== in them (i.e. create clean text), save the file, commit the change, and you're good to go.

Pull Requests

Merging branches (as we just did) can also be done in `GitHub` by creating a *Pull Request*. When you create a pull request in `GitHub`, there's a text box for (optionally) describing the changes that you're suggesting should be merged into the master branch. Once submitted, a Pull Request initiates a process in which `GitHub` compares the two branches you selected (using `diff`) but then, regardless of whether there are conflicts, offers your collaborators (to whom you can assign the task of reviewing) the opportunity to add comments or request additional changes to your feature code before the merge is performed

Note that you can initiate a Pull Request at any point during your coding process, even if all you want to do is share general ideas or a sketched-out pseudo-code version of what you're working on, or when you're stuck and need an additional brain to think about the problem. Further, note that you can continue to commit additional changes and push to the feature branch after having initiated a Pull Request. `GitHub` will include these additional commits in the Pull Request. That way you can go back-and-forth with your collaborators (or yourself), commenting and making revisions, until you're ready to merge.

Clones vs. Forks

Cloning your project is what you did at the very start of this class when you created your repository on `GitHub`, copied the url link to it, and then provided that link to `RStudio`, your `Git GUI`, or `Git` itself to initiate the copying of everything that was on `GitHub` onto your hard-drive. You could also have gone the other way: that is, you could have created the `Git` repository on your computer and then cloned it to `GitHub`. Therefore you can work on your project on multiple computers simply by cloning the repository to each of them. (Just remember to commit and push what you've accomplished in order to pull and keep working on it on your second computer.) You are the owner of that repository. Only you can pull and push to it unless you've granted permission for a collaborator to do so[14]. In fact, whether public or private, you and your permission-granted collaborators are the only ones that can do anything within the repository (such as edit files or create branches).

Forks are in many ways similar but are different from branches, and serve a different purpose. Anyone can fork a public repository. When you fork someone's respository you create a copy of their repository over which you have control. A reason for doing that might be wanting to contribute to someone else's project (e.g., a consortium of biologists all contributing to the maintenance of a database, or a team of R-package developers). Or it might be that you want to use someone's project (or even one of your own old projects) as a starting point for your own new project. What forking does is maintain a connection between the parent repository and the descendant repository (as opposed to simply copying files, the way you would do without `GitHub`). This permits the added convenience of being able to submit Pull

[14]Go to the repository's *Settings*>*Manage Access* in `GitHub`.

Requests between forks, either to offer your improvements or additions to the parent repository, or to pull in added improvements or bug fixes in the parent that were identified subsequent to your having created the fork. Just as for branches, you issue a Pull Request using the "Compare and Pull Request" button, but this time you'll compare across forks rather than your internal branches.

Going back in time

We've already talked about how easy it is to use `diff` to compare back to prior commits, or to compare any two commits in your repository's history. But what if you want to go back in time to one of those previous commits?

Revert

The safest way to undo things is to use `revert`. This won't delete the erroneous commit(s) but rather will create a new commit that reverses whatever the last commit(s) did. This is a good way to keep a full and clean (and linear) history of your branch. In command-line, use `git revert` *`commitID`*. In most GUIs, you should be able to right-click on the commit you want to go back to and select `Revert` or `Reverse commit`.

Reset

The reset function is a more dramatic way to undo commits. It is *not* a good idea to use `reset` if you've already pushed the commits to your remote on `GitHub`. One reason is that you'll create problem if your collaborator has already pulled and based their work on the commits that `reset` removed. That's because `reset` erases the removed commits from your local history. Another reason is that `GitHub` will also not play as nice. The remote on `GitHub` will still have those locally-removed commits, so your local will end up being *behind* the remote. That means that you'll be prompted to pull them back to your local in a never ending loop (unless something else is done[15]).

[15]That something is to *force* the remote to reset by using `git push -f <remote> <branch-name>` when you push after using `reset`.

If you're confident that you want to `reset`, you have a few options: Using command-line, to undo the last commit completely, use `git reset --hard HEAD^`. Changes that were made after the commit-before-last will be lost forever. To undo the last commit but keep your subsequently-changed files in the *unstaged* state, use `git reset HEAD^`. Changes that were made after the commit-before-last will remain but will be unstaged. To instead keep the changed files in the *staged* state, use `git reset --soft HEAD^`. There are many additional options too, including the option to revert just a single file. (See https://git-scm.com/docs/git-reset for more.) To revert to any older commit you'll need its ID number.

You can do the same things in your GUI by right-clicking on the commit you'd like to revert to (not necessarily the commit-before-last) and selecting `Reset master to this commit`. For the latter you'll then be presented with three options: *hard*, *mixed*, or *soft*. *hard* will cause all changes that were made after the selected commit to be lost forever. *mixed* (the default) will keep all changed files in the *unstaged* state. *soft* will keep all changed files in the *staged* state.

Recapturing older commits

You may at some point realize that you had something in a previous commit that's worth rescuing from the past. But you may not want to discard or go through the trouble of reintegrating all the subsequent work you did, as may be necessary when using `reset`. There are a few workflows that people use to retrieve the past. For example, to retrieve specific files or directories from an old commit, use `git checkout` *`commitID filename1 dir1 dir2`* then commit, but note that this will overwrite the current copies of the files (so best to be on a branch). To retrieve a whole commit, *checkout* the old commit in question, create a new *branch* off the old commit, then *merge* back into your master commit and resolve the conflicts in the process. The processes are equivalent in your GUI when you right-click on the old commit or files. Notice that in the latter method lies an important reason for the `Git` motto, *commit early, commit often.* The more frequently you commit (and the better you are at committing good "units" of change), the more targeted (and cleaner) your strikes to the past can be.

The Nuclear Option

One thing to recognize when you're using `GitHub`: Your last push to `GitHub` is your final recovery scenario in that you can always delete your entire local repository and re-clone (Fig. 4). Sometimes you end up messing things up enough to make that the easiest option.

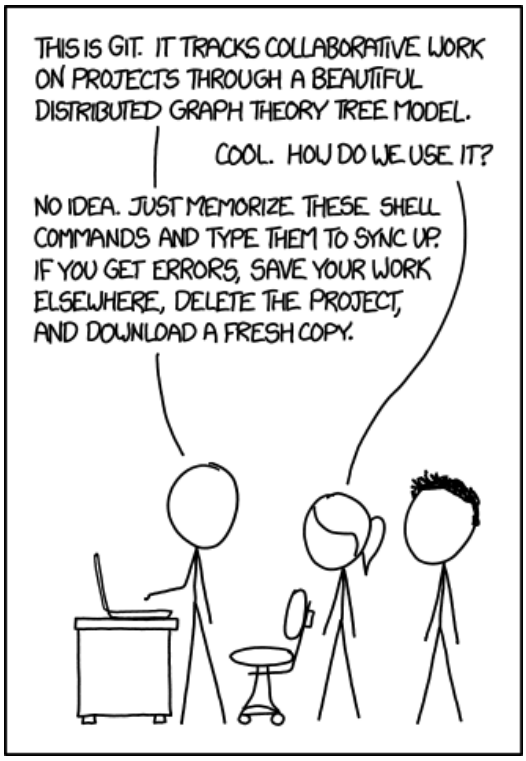

Figure 4: Solving errors in Git (source: https://xkcd.com/1597/)

Project Management with `GitHub`

Everyone has their own ways of managing their life, keeping track of To-Do's, etc. Not surprisingly, that's true for research projects as well. However, when you're already using `Git` to keep track of the contents of your project, its tight integration with `GitHub` enables a few nice conveniences to keep track of both the project and its management in one place. These features are

useful not only in collaborative contexts but also when working solo. We'll touch on only two of them, and only by means of a bare bones introduction. Other features include the ability to host project-specific websites[16] and to create Wiki pages (which you could use as a lab notebook or to create a user manual).

Issue tracking

By now you've undoubtedly already seen `GitHub`'s issue-tracking interface. The basics are pretty simple. Create a short, specific, informative and usefully-unique subject header. Be concise and clear in your subsequent description of the issue. You can use `Markdown` to format your comment, and use *Preview* to make sure it looks right before posting. When your issue is related to a previously-posted issue (even a closed one), link to it using *#issueID*. When applicable, tag your issue with a label. You can add to or edit the default labels to make them more useful for specific project. By assigning issues or using *@mentions* in your comment, others will receive a notification (or email, if they're set up to receive them in their notification preferences). When you close an issue, it's wise to reference the commit in which the fix was enacted. You can always reopen an issue. To see all closed issues in the list of issues, simply delete the default contents of the filter search (i.e. delete *"is:open"*). If you've already setup a *Project board* (see next), you can also assign the issue to it.

Project boards

There are a number of stand-alone apps that provide so-called Kanban boards for project management. The most basic setup entails three columns: *To do*, *In Progress*, and *Done*. When you have a new task that needs doing, you add a card to the *To do* column, then move it over to the next column as your actions on it progress.

In `GitHub` you can have multiple project boards per repository and use them for anything you'd use a stand-alone Kanban app for. In my mind the motivation to use `GitHub`'s project board feature is that (1) everything to do with your project is in one place (major downside being it's only online), and that (2) it integrates nicely with Issues. Assigning an issue to a project

[16] You can even use `GitHub` to host your own website for free!

board will cause it to appear in the “To do” column. Closing the issue will move it to the “Done” column. You can change the settings of your project board so that any newly-created issue is automatically added to the “To do” column.