More with `Git` and `GitHub`

October 13, 2020

Contents

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. The goal for today is to extend our workflow by making it more flexible and adaptable to more diverse project structures and more general 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.

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). 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).[2] 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 additonal information in them for interpreting, presenting and otherwise doing things (e.g., calculations) with your data and text that requires specialized software.[3] 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;

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

[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] See back to our `Structured Projects` class.

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

new way;

4. every time you open and save a `.xls` or `.doc` file, a whole bunch of that computer language information is changed, causing `Git` to unnecessarily track these changes;

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.[4] Microsoft- and Apple suite files and the like are (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 you don't want `Git` to track (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`).

Branching

So far, when we've committed to our `Git` repository, we've been doing so to its *master* branch. The *master* branch has been our only branch, and

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

every change we've made to a file has been changed sequentially (i.e. has overwritten what was there before). In other words, our commit workflow has been pretty much 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[5]), or you might want to try writing a section or paragraph of your manuscript a different way[6]. 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 often 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[7]. You can also do it within `GitHub`. To create a branch using command-line, type `$git checkout -b` *`branchname`*. `Git` will not duplicate any of your files, but it will keep track of all changes made within the branch (when you commit them to the branch). If you add, edit, or remove files (and commit) while your on your feature branch, those changes will not appear in your master branch (and vice versa). Thus, if you were to add a file while in your feature branch and were then to switch back to the master branch (using your `Git` GUI, RStudio, or `$git checkout master`[8]), you wouldn't see that new file 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, one each for testing out changes to each of several scripts or manuscript sections. Only once 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

[5]See `Faster Computing` class later.

[6]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.

[7]Unfortunately not in `RStudio`.

[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.

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]. Then, in collaborative contexts, it's often useful to add one more 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.)

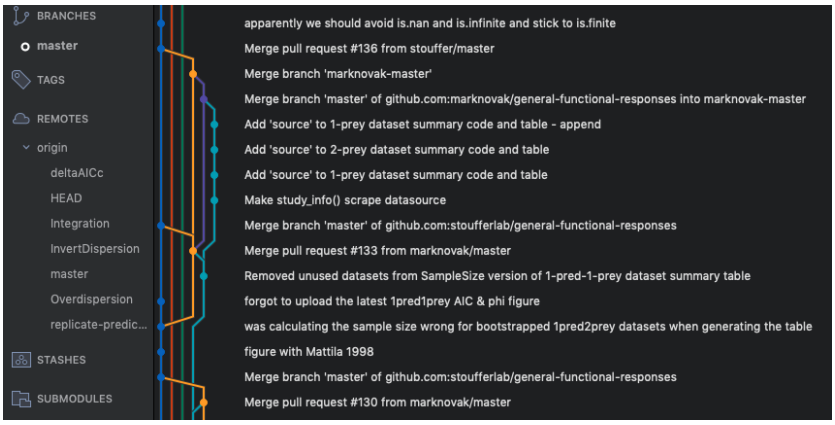

Figure 2: `SourceTree`'s branch visualization.

[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.

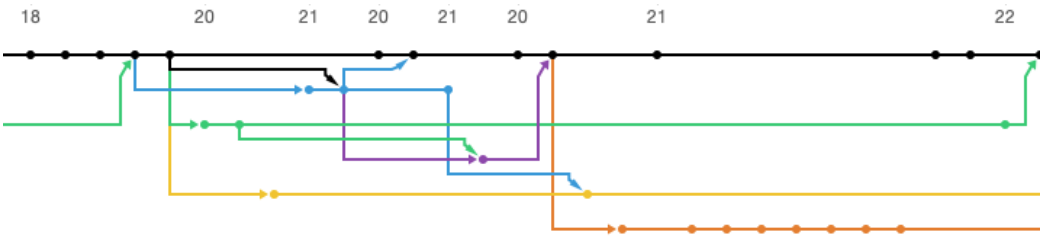

Figure 3: `GitHub`'s "Network" branch visualization.

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 (in the same branch). 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 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.

[10] When you start using LaTeX, notice 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.

`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[11].

Now, 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 everthing 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. An alternative that solves this problem is *rebasing*, but that comes with other potential challenges.

When there are no conflicts, `Git` will simply amalgamate[12] 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`***).

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

[11] 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 isn't a fundamental problem).

[12] 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.[13]

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. 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

[13] Admittedly, this *can sometimes* be a little annoying until you get the hang of it.

add comments or request additional changes to your feature code before the merge is performed[14].

Note that you can initiate a Pull Request at any point during your coding process, even if all you want to do is share screenshots or general ideas, or when you're stuck and need an additional brain to think about the problem. Also, note that you can also continue to commit additional changes and push to the feature branch even after having initiated a Pull Request for it. `GitHub` will show include these additional commits in the Pull Request view, so you can go back-and-forth with your collaborators, commenting and making revisions, until you're ready to merge.

Forks vs. clones & branches

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: create the `Git` repository on your hard-drive and then clone it to `GitHub`. You can work on your project from multiple computers by cloning the repository to them. (Just remember to commit and push what you're working on on the first computer 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 so[15]. In fact, whether public or private, you and your permission-granted collaborators are the only one that can do anything to the repository (such as 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

[14]Note that you can create a Pull Request and perform the merge yourself; you don't need to have a collaborator to use `GitHub` for merging.

[15]Go to the repository's *Settings ¿ Manage Access* `GitHub`

is maintain a connection between the parent repository and the descendent 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 Requests between forks, either to offer your improvements or additions to the parent repository, or to pull in added improvements or fixes to bugs 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 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 revert to a previous commit?

Undoing the last commit

If it's only your last commit that was made in error and you want to undo it, you have a few options:

To undo the last commit completely, use `git reset --hard HEAD^`. This rolls back your repository to the previous commit; changes not reflected in the commit-before-last will be lost forever.

To undo the commit but leave the files in that state but *unstaged*, use `git reset HEAD^`. This rolls back your repository to the previous commit, with any changes not reflected in the commit-before-last remaining. The last commit will be undone and nothing will be staged.

To undo the last commit but leave the files in that state and *staged*, use `git reset --soft HEAD^`. This rolls back your repository to the previous commit, with any changes not reflected in the commit-before-last remaining as staged changes.

Finally, if you just want to make a small change to the last commit you made (e.g., you forgot to include a file, had a minor typo somewhere, or want to change the commit message), you can `amend` the last commit without creating a new commit using `git commit --amend -m "New commit message"`. Be careful though as this is changing history! Also, once you push a commit to GitHub, you cannot amend it. Doing so will create an error the next time you try and push to GitHub.

The last push is your worst case recovery scenario (see Fig. 4): when all else fails, you can always delete your entire local repository and re-clone from GitHub to pick up at the last pushed commit.

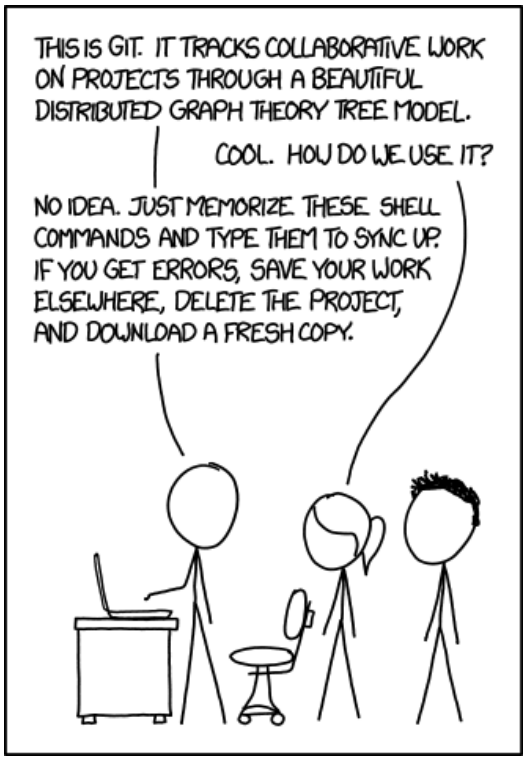

Figure 4: Solving errors in Git (source: https://xkcd.com/1597/)

Cherry picking

Try out before discussing: branch off old commit, change single file, merge into last commit

Project Management

Issue tracking

Edit your labels (useful for personel project management too).

By assigning issues or using @mentions and references inside of Issues, others will receive a notifcation (or email, if set up in their notification preferences).

Open vs. closed - how to use search field

Project boards

Other features

There's a bunch of other features on GitHub that we won't get into (such as Wiki pages,)