Github for Social Scientists[*]

Mehmet Seflek

October 10, 2017

1 Introduction

Github is a fantastic platform for collaborating on coding tasks. It does, however, seem significantly more "clunky" than other sharing and collaboration tools like Dropbox. This is on purpose: given how fickle programming can be, it is crucial to ensure that in a collaborative environment decisions to incorporate changes into your "deployed" program are deliberate and traceable. This not only ensures that mistakes that can potentially break your code (or worse, change your results unintentionally) are caught early, but also helps to track the changes and trace who owns them.

One (highly simplified) way to think about Github versus other tools like Dropbox is to compare the "Track Changes" feature of Microsoft Word to saving edited versions of a document:

- With Microsoft Word, edits and comments made by someone else must be deliberately incorporated into the main version of the document by accepting each change and reacting to each comment. Since each individual editor and change is labeled within the document on a very fine scale (ie. letter by letter), this makes identifying and incorporating changes easy. Working simultaneously on the file with multiple collaborators, however, becomes difficult (though Google Drive certainly helps with this) and Word is not typically used for coding.

- Alternatively, one could save versions of a document as it is edited, where the latest version is the deployed version of the program. Each collaborator makes edits to the file and either saves his/her own versions that is eventually incorporated, or they edit the latest version. Tracking changes by each editor can be difficult since they are not clearly labeled. This can get messy quickly, as tracking changes and versions gets complicated (I have a file in my DropBox titled "Analysis_v3.1_SN_1.do"; it would take me one hour to figure out where it fits in my workflow).

[*] This guide is a living document and will be updated as necessary.

While these metaphors do not perfectly fit Github, it is useful to think of Github like Microsoft Word's track changes feature. In effect when using Github for coding in a collaborative way, it allows you to:

- Simultaneously collaborate on developing code without breaking or inadvertently changing existing code
- Identify changes from collaborators throughout the development of the code
- Develop new "features" or "sub-analyses" in a gated environment ("branches") without affecting the running code
- Actively and deliberately manage code versions both across collaborators and time

2 Why should I care about this in the social sciences?

Well, this is a good question answered by the correct answer to everything in economics: it depends.

If a research project requires many collaborators with potentially multiple individuals coding, then Github allows you to manage the coding process without having to develop your own internal code sharing, versioning, and merging procedures. In other words, there is no need to reinvent the wheel. For a field experiment, for example, coding happens at multiple levels (ie. sampling design, quality control, cleaning, analysis, etc) and with multiple people in the pipeline. This codebase usually needs to be managed at each level as well as centrally to ensure each piece fits together. In addition, its also important to track versions so that a future version of the team can easily retrace the analysis that has been done.

Even if you are writing code by yourself, you actually *are* collaborating with someone: your future or past self. Using Github to manage the coding process allows your to track the changes to your code over time easily, and revert or partially revert your changes seamlessly. It also forces you to make deliberate choices in code management. While this may sound annoying at first, having a structure that you commit to pays dividends in the future when you have to fish out that piece of code that you thought you didn't need.

3 Rough guide to working with repositories

[Note: I initially wrote this to share with undergraduate RAs, so I am presuming a certain knowledge of general Github workings. I also assume that you've successfully got it installed on your system. Perhaps I will extend this at some point to include those steps.]

Like I mentioned above, the action of "syncing" code is very deliberately done using Github. I will take a step by step approach to doing this below, *assuming that there is* *a repository that you are a collaborater on.* All of the steps below are run in a terminal session in the directory you wish to work in. Note that

3.1 Creating a repository

TBD

3.2 Syncing repository to your local machine

To sync the repository to your local machine, use the following command, replacing `[your-url]` with the URL of the repository with `.git` added to the end.

```
git clone [your-url]
```

Now you should have all of the files associated with the repository on your local machine. This does not mean that any changes you make will automatically be synced back to Github, however. This requires a bit more work.

Suppose that after you have “cloned” (now I will start introducing Github specific terms) the repository, you want to get to work on introducing a new feature to the existing code. Now version control becomes important. Github uses a concept called “branches,” which are copies of the code that can be worked on without affecting the “master branch,” or the version of the code that is the latest working copy. This allows you to work on developing additional features of analyses without potentially breaking running code. To do this, we create a new branch named `tinkering` (or whatever name you want to use):

```
git checkout -b tinkering
```

The `checkout` command essentially does what you may have visualized. Like checking out a book from a library, `checkout` activates a given branch. The `-b` flag essentially says “create this branch if it does not exist, and activate it.” You can activate a different existing branch by dropping the `-b` flag.

Now, suppose that you are now in the `tinkering` branch, and you make some changes, and now you want to incorporate your code into the master branch. First, verify that you are in the `tinkering` branch by typing the following:

```
git status
```

You should get something like this:

```
On branch tinkering
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

Now, things get interesting. To “stage” your changes for potential incorporation into the working code, you have to be explicit that you want to do this. You stage the changed files for incorporation by the following command:

```
git add -A
```

The `-A` flag says to stage all files that have changed within the repository. You could be very specific and list a filename instead of the `-A` flag.

We’re almost there. Now, we have to “commit” our changes to essentially create a checkpoint. This might seem weird – didn’t we just do this with the `add` command? Not exactly: `add` flagged which files will be saved in the checkpoint, but committing actually *saves* that checkpoint.

So now we make our first commit!

```
git commit -m "First commit"
```

Now we’ve created a checkpoint! You can create as many checkpoints as you’d like, but it’s up to you to make these checkpoints. To beat a dead horse, there is no “automatic syncing”.

Now you might start to get annoyed, because your changes are still not in the repository. You now have to “push” your changes to the repository:

```
git push -u origin tinkering
```

Here, we are sending the `tinkering` branch to the repository. Here `origin` is an alias for the remote repository that we’re working with – this is copied over when we cloned the repository.

Lastly, if you want to resync the latest version of the repository to your local repository, just use the following:

```
git pull
```