**Version Control with Git and GitHub**

**Biogeocheminar – Jan 28, 2016**

**Overview**

Today we are going to learn how to use Git to store a detailed, organized record of our coding projects. This record can be used to restore code to previous states, and to partition it into different “branches” for testing or for specialized application. What we’ll do today will only scratch the surface, and we’re not even going to touch the aspect of Git that it is best known for – that is, streamlining collaborative coding projects. If you’d like to learn how to do that, there are official workshops offered through UW, as well as many online tutorials.

Here’s the basic workflow we’ll be using:



**Setup and boring stuff**

*yak shaving* (idiom): Carrying out a series of uninteresting, preliminary tasks before addressing the thing you set out to accomplish. Apparently from the show “Ren and Stimpy”.

**Make Github account**

0. Make sure you have Bash and Git installed. If you're on OS X or Linux, you probably have both by default. If you're on Windows, you can access Bash (a command-line shell) and Git through [Cygwin](https://www.cygwin.com/).

1. Go to [www.github.com](http://www.github.com/)

2. Fill out account info and keep track of your username and password for later.

3. Click green “Finish sign up” button. The only advantage of the paid version is that you can make your submissions private. I’ve heard we all have free access to this privilege as UW people.

4. Check email to verify account.

**Make a remote repository**

This will make a folder on Github where your code and files will be stored. (Note that I’m going to use the words “folder” and “directory” interchangeably. A repository is a *special* folder that keeps track of stuff for you.

5. Click green button that says “new repository.”

6. Call the repository “testrepo” and choose “Initialize this repository with a README.” Below that, click “Add .gitignore” and choose “R” for this example. We’ll talk about these things later. Now you can create the repository with the green button.

**Configure Git on your computer**

7. Open Terminal or Cygwin (Wait for it to display your username and computer name.).

8. Tell Git your name with:

git config --global user.name “your name”

9. Now tell it your email addess with:

git config --global user.email “your email”

10. Tell Git to automatically handle Unix vs. Windows end-of-line differences with:

git config --global core.autocrlf input

on Linux or OS X, and:

git config --global core.autocrlf true

on Windows.

**Command-line crash course**

The standard language for command-line operations on Mac and Linux is called Bourne Again Shell, or Bash (so called because it improves upon the Bourne Shell). Here are the core commands in Bash:

|  |  |  |  |
| --- | --- | --- | --- |
| **Command** | **Purpose** | **Example 1** | **Example 2** |
| pwd | Print working directory (where you are now) | pwd |  |
| cd | Change directory | cd Desktop | cd /home/mike/Downloads |
| ls | List directory contents | ls | ls –a (to include hidden files) |
| mkdir | Make directory | mkdir folder1 | mkdir folder1 folder2 |
| touch | Make file | touch file1.txt | touch file1.txt file2.csv |
| rmdir | Remove directory | rmdir folder1 | rmdir folder1 folder2 |
| rm | Remove file | rm file1.txt | rm –rf folder1 (WARNING: will delete any file or folder and all contents) |
| CTRL/CMD + C | Abort whatever is happening |  |  |

For the first couple of steps, we’ll be messing around with a few of these commands, and there will be a lot of system-specific tedium. After a while, though, we’ll be in Git and everybody will be using the same commands. I’m only going to differentiate between Windows and OS X, in part because I don’t think there are any Linux users in the house, and if there are, I assume they can fend for themselves when it comes to using Bash.

**CAUTION**: If you decide to explore the command line more on your own, just be aware that it’s possible to really mess up your computer. However, Terminal commands are like wild mushrooms – there are only a few that are dangerous, and they’re easily identifiable with some experience. At the beginning, though, you’ll want to be very careful. Definitely don’t use scripts carelessly that you find on the web, just like you wouldn’t eat a handful of mushrooms offered to you by a strange person in the forest.

**Cloning your remote repository (i.e. making a local copy)**

10. Start by creating a new folder on the desktop. Call it “test\_projects”.

11. Copy the filepath of that folder to the system clipboard:

Windows 7 or 8: hold shift and right-click the folder. Choose “copy as path”.

Windows 10: open the folder’s location in your file manager. Choose “Copy path” in the Home tab.

OS X: click the folder and press COMMAND+I. Highlight + copy the path listed beside “Where:”.

12. Navigate to that folder in Terminal by typing “cd”, then a space, then pasting the filepath, then pressing enter. On my system this looks like:

cd “C:\Users\Mike\Desktop\test\_projects”

On OS X, you’ll have to manually type the name of the folder after the path. It might look like:

cd /Users/Mike/Desktop/test\_projects

**Note** that you can’t by default paste into the Terminal with the usual keyboard shortcut, so just right click and select paste. If you want to find your system-specific copy and paste shortcuts, Google can tell you what they are. It can also tell you how to change them to the shortcuts you’re used to. In fact, it can tell you how to customize your Terminal (and therefore your whole computer) in just about any way imaginable. Welcome to Narnia.

13. Now you’re in the new folder. Cygwin shows you this by listing the filepath after your name. On Mac, try the “pwd” command to see your working directory.

14. Now it’s Git time. Go back to Github and copy the URL listed a couple of inches to the right of the green button. This is the web address of your remote repository. Go back to Terminal and enter:

git clone <paste that url you just copied, without ‘<>’>

This will copy the exact contents of the repository you made on the web into a folder on your machine.

15. Try out the “ls” command now, and it should return the name “testrepo”. That means it found a folder called “testrepo” inside the folder you’re currently in, which is “test\_projects”. If you’d like to visualize this in the usual way, go ahead and open your file manager (Finder or Windows Explorer) and navigate to these folders.

**Gitting real**

Okay, now we’re going to start learning some useful stuff, and most of it is going to be within Git itself. Git commands are entered into the Terminal just like the commands we’ve been using. The only difference is that every Git command is preceded by “git” and a space. In this section we’re going to learn to save or “commit” work to our local repository, create separate branches of that repository for working on different things, look through our revision history, and more.

**Committing to the local repo, pushing commits to the remote repo**

1. “cd” into the repository (the folder called “testrepo”) and enter:

ls –a

This will show you everything in the repository, including “hidden” files that your system might normally not show you. You’ll see a folder called “.git”, as well as the two files we chose to include when we made this repository on GitHub. You may never have to mess with the “.git” folder, but just know that this is where all the magic happens. The README file is just for writing instructions on how to use code that you submit to GitHub, in case somebody else comes across it and wants to use it for their own work. For our purposes, READMEs are probably unnecessary, but it’s good practice to include them in projects intended for others. We’ll talk about the “.gitignore” file at the end of this section.

2. Type:

git

to see a list of common commands. If you want more info on any particular command, type “git help <command name>”. You’ll notice these help files can be very arcane though, and Google is almost always a better teacher.

3. Type:

git status

and you’ll see that you are on the local “master” branch (which is the only branch at the moment), that this master branch is up-to-date with its counterpart on GitHub (“origin/master”), and that there’s nothing new in the working directory.

4. Let’s make a fake code file in our repository. You can do this in the Terminal with the “touch” and “echo” commands, but for the sake of keeping this document short, I’m just going to assume we’re all using our regular file managers. Call the document “code.txt”. Then use Notepad or TextEdit to write a line into it. Let’s all make that line “here is a line of code”, without quotes, for consistency. Add an empty line after it. This empty line at the end of a script is good practice, and it will simplify something for us later. Save the file.

5. Now go back to the terminal and again retrieve Git’s status. The master branch is still up-to-date with the origin, because you haven’t committed any changes yet, but you’ll see that your new code has been detected as an “untracked file.” Type:

git add code.txt

to “stage” your changes. This puts them in a kind of limbo file, so that you can review what you’ve changed before saving or “committing” it to the local repository. Two side-notes: if you press TAB after typing the first few letters of a command or filename, and there’s no ambiguity about the word you’re looking for, Terminal will auto-complete it for you. Also,

git add .

will stage all files at once.

6. Type:

git commit –m “this is my first commit message”

Now you have saved your changes to your local repository. You cannot commit changes without including a message. In real life, your message might be something like “changed figure 3 to a scatterplot”. Note that if you try to commit without a message (i.e. just type “git commit”), you may be transported to an ancient and beautiful text editor called Vi or Vim. Just press ESC, then :, then q, then ENTER, and you will be returned to the main Terminal, where you can re-enter the command correctly.

7. If you check the status again, it will tell you that your local repo is ahead of the remote one by one commit. Now it’s time to sync with GitHub. Type:

git push origin master

This means you are sending (pushing) your changes to the repository on GitHub (named “origin” by default) from the local repository (named “master” by default). You will be prompted for the username and password you made on GitHub. To use Git solely for remote file backup, you now have all the commands you need. You can check the status again and see that everything is clean. Pushing to the remote repo can be done at any time, assuming your working directory is clean.

**Checking out old commits**

Say you broke your code, and now you can’t get anything to work. Don’t spend an hour debugging. Just go back to an older commit where everything is still working! Let’s see how to do that.

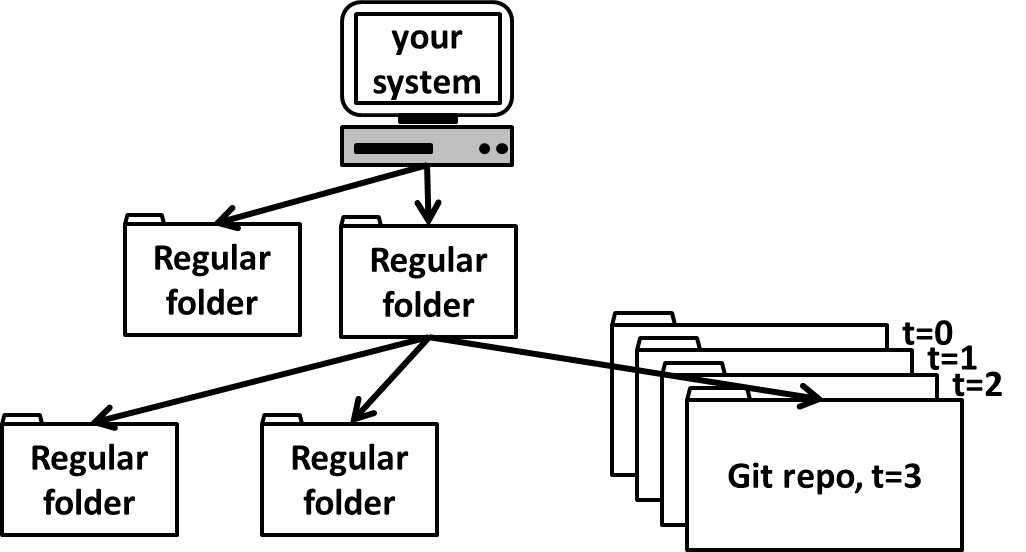
8. Go back into code.txt and add a line that says “here is a modification to my code”, with an empty line after it. Save and close the file. Open Excel and put numbers in a few cells, then save the spreadsheet in your repository as a CSV called “newfile”. Commit changes with a new commit message.

9. Arrange your windows so that both Terminal and the file manager are visible. See your CSV file there? Now type:

git checkout HEAD~1

to check out your previous commit. Look at your CSV and – *OMG IT’S GONE*! No… it’s not gone, it’s just… **in the future!** **Dun- dun- DUUUUUN.**

Hopefully now you can see the power of this framework. Essentially, where you once had a folder full of code files, data files, helper files, other folders, etc., each packed with separate sections for separate purposes and giant blocks of commented-out text that you’ll probably never use again but that you just can’t bring yourself to delete, **now** you have a *time-series* of folder snapshots that you can move between. Here’s how I visualize it:



**The .gitignore file**

You will eventually have things in your repository that you don’t want Git to keep track of. These would be things like system files, Word documents, images, etc. When we created this repository on GitHub, we added a .gitignore that was tailored for R projects. That means it’s already set up to ignore .Rhistory files, session data, and some other stuff.

If you can’t see the .gitignore in your file manager, that’s because your computer is hiding it from you. You can find out how to show hidden files and folders by searching “show hidden files and folders <your OS and its version number>” .

The .gitignore can be edited in any text editor, but your computer might not know how to open it. One solution is to open it from the command line, e.g.:

Windows: notepad .gitignore

Mac: open –t .gitignore

If it’s all on one line when it opens, that’s because Notepad is lame. You can get a better text editor like Notepad++, or just edit from the command-line with:

echo “line to append to the file goes here” >> .gitignore

To ignore particular file extensions, add lines like this:

\*.docx

\*.png

To ignore a subdirectory, add a line like this:

/folder\_name

**Working with branches (only if you’re feeling intrepid)**

1. Read the output from your last command and you’ll see that you can make changes to this old commit and either let them float away into the ether or save them to a separate branch. Let’s see how to do that. Type:

git log

to see a log of the commits you’ve made. You’ll see the one that was made when you first cloned the remote repository, and you’ll see the first commit you made. The second commit is not there, because you’re still in the past. Go back to your most recent commit by checking out the branch you’re on:

git checkout master

And then view the log again.

**Creating branches**

2. Now let’s make a new branch and call it “newbranch”:

git checkout –b newbranch

To see which branch you’re on at any time, type:

git branch

Add a third line to your code file that says “this was added in a new branch”, followed by a blank line. Then add and commit the change.

**Comparing branches**

3. Type:

git diff master newbranch

to see the differences between the branches from the perspective of newbranch. Here’s what the output means (Ignore lines 1-5 unless you’re interested):

LINE 1: a/fileA and b/fileB are the two files being compared. In this case they are the same file.

LINE 2: codes Git uses for the version of the file before and after changes. Not important now.

LINES 3 and 4: source (---) and target (+++) of the changes being made. One of them would be /dev/null if you had just created or deleted the file.

Line 5: @@ from-file-range to-file-range @@ [header]

Where from-file-range is in the format -<start line>,<number of lines>

And to-file-range is in the format +<start line>,<number of lines>

FOLLOWING LINES: This is where it actually lists the changes. Stuff with a plus was added and stuff with a minus was removed.

**Merging branches**

4. Say you had a master branch with your working code, and a testing branch where you were trying out new things. If you got something working well on the testing branch, you might want to combine the branches again to simplify things. Go back to the master branch with :

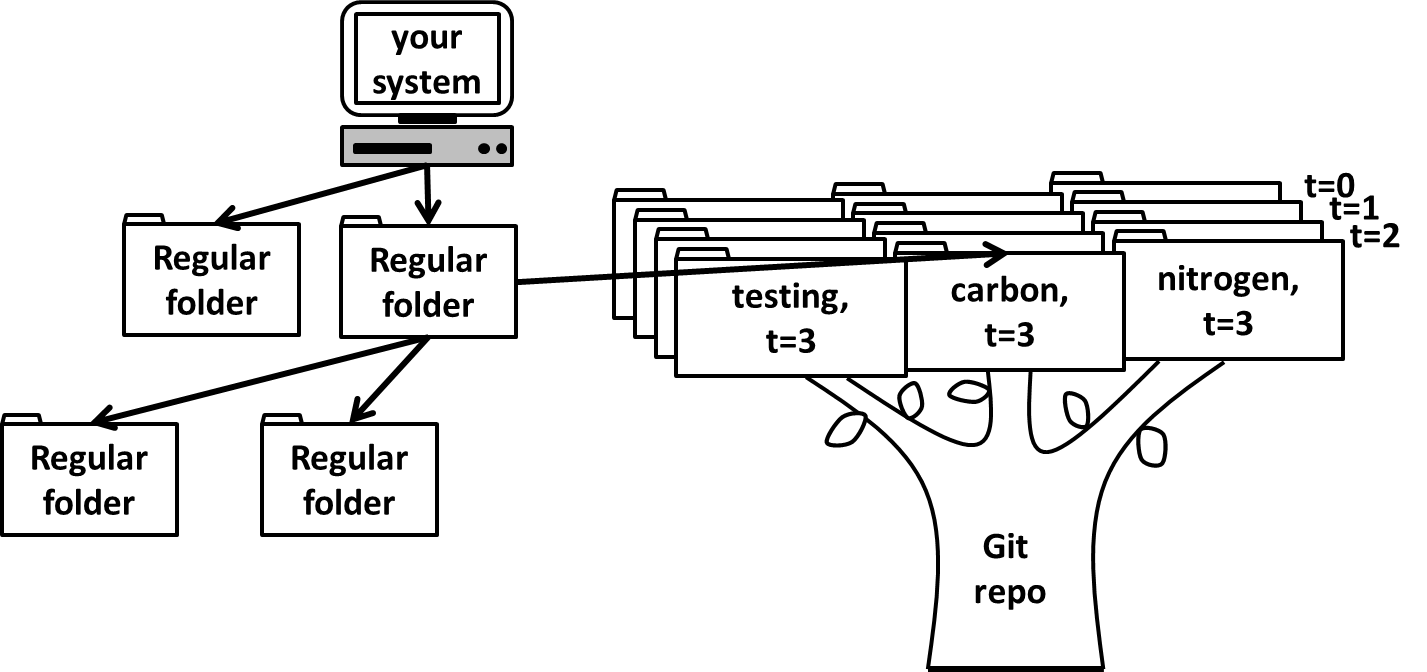
git checkout master

and then merge the two branches with:

git merge newbranch

Since all we did was append a line to the file, this merger is handled automatically. If there were conflicts where the same line differed between branches, you’d have to reconcile the difference manually before merging.

Now you have even more power. Not only can you go back to any previous snapshot of your repository, but you can jump between different branches that cater to different needs. Essentially, where a non-Git-user has to figure out how to organize their code in a tree of nested folders, or as a big, messy pile in one folder, you have a whole *matrix* of organizational real-estate, spanning time and space, at your disposal. It’s still just one folder on your computer, but *effectively* it’s like this:



Not only that, but the whole structure is backed up online, in a giant, elegantly designed, collaborative utopia. By the way, if you ever want to install an R package from GitHub, the R code for that is below. There are some kickass packages that aren’t available through CRAN.

install.packages(“devtools”)

library(devtools)

install\_github(“author/package”, subdir = “subdir\_name”)

Also, you can turn any local directory into a Git repository and then link it to GitHub with:

git init

git remote add origin <URL of a remote repo>

Now, git working! ^\_^