Version control with Git and GitHub

A good workflow should save you time, make it easier to find and fix errors, clearly document your analysis steps, save your work somewhere safe, keep your work history (in the way that "Track Changes" does), and make it easy to update and rerun your analysis, for example if more data becomes available. When done correctly it will bring a deep sense of inner calm from knowing that everything is in its right place.

The workflow we'll use for much of this course is based on a combination of the following three tools:

- **Git** for version control, sharing code, and communicating code and associated results;
- **R Projects** for linking your work in R (code and narrative text) to GitHub;
- **Quarto** as a way of combining code and narrative text, and more generally for communicating your work to the outside world.

Each of these can be used in their own right, but their real value comes from using them together. Quarto was covered in the last lecture; here we focus on Git and R Projects. R Projects are pretty simple, so we start with them.

1 Git and GitHub

Git can do a lot of things, and this complexity can be overwhelming at first. I struggled (several times) to get a working understanding of version control with Git, so I've written this as the kind of very simple "how to" guide I would have liked to have had. If you already know how to use Git, or are more computer savvy than me, you might find it too elementary – skip ahead as you see fit.

Here are two things to keep in mind when getting started:

1. *Git*, *GitHub*, and a *Git client* go together. You need to know how to use them all (unless you plan on using the command line exclusively).
2. Most of what we'll do requires only the basics of Git. You can pick up the rest later.

Let's first look at how *Git*, *GitHub*, and a *Git client* fit together:

Git: Git is a version control system – a way of keeping track of changes to a file over time (like "track changes" in Microsoft Word, for example). Originally used for source code, Git is now used by data scientists to track a set of files associated with a project over time. Git calls the set of files a *repository*, and it can contain data files, code, documents.

GitHub: Git is a program that lives on your computer. You could, if you wanted to, only ever use it locally to back-up your files. Most people though, use Git as a way to collaborate on projects, and share code and associated files. That needs some way of hosting Git-based projects on the internet, which is where *GitHub* comes in. GitHub is a web platform that hosts/stores Git-based projects. There are other hosting services out there, but we'll just focus on GitHub.

Git client: Git can either be run through the command line or using a helper application like a GUI that assists with the process. The helper application is called a client. Some users prefer the command line, and indeed there are some things that can only be done through the command line, but when getting started the use of a client can be a huge help. The good news is that RStudio comes with an inbuilt Git client, which you can use whenever you set up an *R Project*. This is enough for most of our needs.

There are two "phases" to using R with GitHub:

1. *Setting up.* This involves installing Git, getting a GitHub account, connecting local Git with GitHub, and connecting RStudio with local Git. This can be a bit of a pain, but the good news is
   - you only need to do this once,

- there are excellent instructions online (see the next section)
- I (or someone else) will provide help to get you set up, if needed.

2. *Working with GitHub.* This basically just involves remembering to do two things:
   - linking any new R projects to a GitHub repository
   - periodically saving your changes ("committing") and uploading these onto GitHub ("pushing")

We'll look at each of these in turn.

1.1 Setting everything up

This is really difficult to do in class – inevitably there will be one or two problems but it is impossible to know what these will be, and troubleshooting them will take up valuable time. So, I've asked you to **please set up your system in advance** using the following steps:

1. Follow the basic instruction steps here. These are the best instructions I've found and most common problems are covered.
2. If you have trouble with any of the steps, send me an email or post to the forum and I (or others) will try to help.

1.2 An intro to working with Git through the command line

We'll mainly use Git through RStudio, which offers a simple Git client. But first we'll introduce Git by using it on the command line. The following is a kind of minimal set of instructions that will showcase some of the basic Git functionality. I don't go into much detail here – the idea is to discuss what's going on in class, and you can repeat the steps here later if you like.

1.2.1 Saving your work: git add, git commit

1. Open the shell/terminal and cd to the directory you want to work in.
2. `git init`. This sets up an (empty) git repository in the folder.
3. `git status`. A useful command to see what files are being tracked.
4. Make a text file with "this is the first line." in it, save to the working directory as *demotext.txt*.
5. `git status`
6. `git add demotext.txt`. Adds *demotext.txt* to the staging area.
7. `git status`. Note status of *demotext.txt* has changed.
8. `git commit -m "added first line"`. Commits with a message (-m).
9. `git status`
10. `git log`. Shows the commit history.
11. Now add a second line of text to *demotext.txt*: "this is the second line."
12. `git status`
13. `git diff demotext.txt`. See changes to *demotext.txt*.
14. `git add demotext.txt`
15. `git status`
16. `git commit -m "added second line"`
17. `git status`
18. `git log`
19. Now add a third line of text to *demotext.txt*: "this is the third line."

1.2.2 Going back in time: git checkout, git revert, git reset

1. Say we don't want the third line, and want to return to the state at the last commit.
2. `git checkout demotext.txt`. Returns *demotext.txt* to state at last commit. If you put a commit id after `git checkout` returns the file to state at that commit.
3. Say we'd committed the change before we realised we didn't want it.
4. Add back the third line of text to *demotext.txt*, save, and add and commit with the message "added third line".
5. `git log --oneline` to get the id of last commit (the one where you added the third line).
6. `git revert <commit_id>`
7. `git log`
8. Open up *demotext.txt* to see the third line is gone.

9. You can also 'revert a revert'.
10. `git log --oneline` and find the last commit id (the revert).
11. `git revert <commit_id>`
12. Check *demotext.txt* to see the third line is back.
13. You can also use git checkout for this.
14. `git log --oneline`, look for commit id for the "added second line" " commit.
15. `git checkout <commit_id> demotext.txt`
16. Check *demotext.txt* to see we're back to having just two lines.
17. Suppose we realise we've made a bit of a mess and want to get rid of the last few commits and just return to the "added second line" commit.
18. `git log --oneline` and find commit you want to return to.
19. `git reset --hard <commit_id>`. This removes intermediate commits, so use with caution!

1.2.3 Trying something new: git branch, git checkout

1. Say you want to try something new on *demotext.txt*. Can do this with branching, which creates a separate branch to work on, without affecting the master branch. Later on you can merge the branch back with the master or delete the branch without merging.
2. `git checkout -b test-feature` (equivalent to `git branch test-feature` plus `git checkout test-feature`)
3. Make some changes to *demotext.txt*. I added a line "some changes I'm not sure will work!".
4. `git add demotext.txt`
5. `git commit -m "testing feature 1"`
6. Check *demotext.txt*, see that feature is there.
7. `git checkout master`. Returns to the master.
8. Check *demotext.txt*, verify its back to the old version without the feature.
9. Add a line "this is the third line" " to *demotext.txt*, add and commit. Remember we're still on the master branch, so this is an update to the master.
10. `git checkout test-feature`. Back to the test branch.
11. `git merge master`. Merge with the master branch.
12. Git tries to merge automatically, but if you get a merge conflict you need to resolve it manually, then add and commit the resolved version.
13. Open *demotext.txt* in a text editor, fix it up, then add and commit.
14. Add a new line to *demotext.txt* "another test feature"", add and commit with message"testing feature 2".
15. `git checkout master`. Returns to master branch.
16. Check *demotext.txt*, see none of the test feature text appears.
17. `git merge test-feature`. Pulls test features into the master by merging with the test branch.
18. Check *demotext.txt* to see test features have been included.
19. `git branch -v`. See the last commits on each branch.
20. `git branch -d test-feature`. Delete the *test-feature* branch.

1.2.4 Going online: git push, git pull

All of what we've done so far has involved local changes. Often you'll want to also save your commits online, in which case you need to "push" your commits to an online hosting platform for Git-based projects (like GitHub). If you want to share your work with others, or work collaboratively, you'll also need GitHub or an equivalent.

1. Create a GitHub repository. Goto https://github.com, log in, find your way to "Repositories", and click the green "New" button.
2. Give your repo a name and description.
3. Don't tick the box "Initialize this repository with a README"
4. Click "Create repository"
5. Copy the URL in the box, which will be something like `https://github.com/iandurbach/myproject.git` (if you're using SSH, this is also where you can get the SSH key and passphrase).
6. Back in the shell/terminal, cd to the directory you've been working in.
7. `git remote add origin <URL>`. "origin" refers to the name of the remote, you can call it whatever you want.
8. `git pull origin master`. Pulls any changes from the master branch of the online repo to your local master. There won't be anything in the online repo yet so nothing will happen (unless you initialized the repo with a README, in which case that will be pulled).

9. `git push origin master`. Pushes your commits to GitHub.
10. Check the appropriate file(s) are on GitHub.

If you ever want to remove a remote, use `git remote rm origin` (remember origin is just the name of the remote, your's might be different). If you want to point the remote to another URL use `git remote set-url origin <newURL>`.

1.3 Working with Git and GitHub through RStudio

In this section we'll set up an R Project and link it to a GitHub repo. That will make it easy to push changes we make in RStudio to GitHub, from within RStudio.

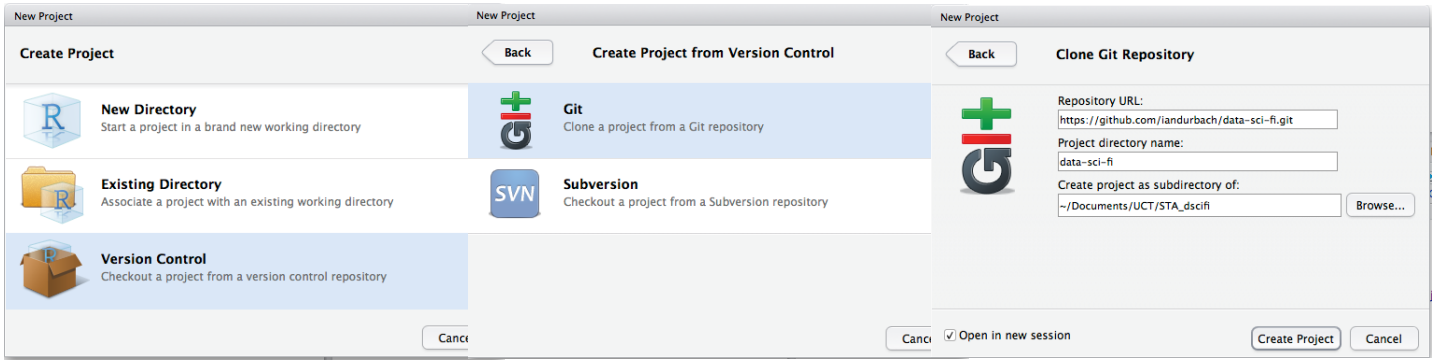
1.3.1 Make a new repository on GitHub

Use the instructions above, except this time tick the box "Initialize this repository with a README". I called my project *data-sci-fi*; you can call yours whatever you like.

1.3.2 Start a new R project in RStudio

We've done this step before, but this time we're going to link the new project to GitHub (we're cloning an empty GitHub repository).

1. Start RStudio
2. Under File, choose New Project and then Version Control and finally Git



(a) Step 1 (a) Step 2 (a) Step 3

3. Paste the URL you copied from GitHub at the end of the previous step into the "Repository URL" box. You can always log back into GitHub and find this URL if you need to.
4. Give the project a directory name and choose where you want to create it. Think about where you want to create the project i.e. put it somewhere on your computer that makes sense, and that you won't want to change later on.
5. Tick the "Open in new session" box
6. Click "Create project"

1.3.3 Do something

In this step you just "do something" you would usually do in R. For example:

1. Use your browser or file manager (e.g. Windows Explorer, Finder) to browse to the project folder you just created. In this folder, create two new folders, one called "data" and another called "output".
2. Go back to RStudio and verify that the new folder and file has been detected.
3. Open a new script file (Cmd/Ctrl + Shift + N) and type the following code in it:

```r
library(ggplot2)

# create some data
x <- runif(100)
# save as RData file
save(x, file = "data/randomnumbers.RData")

# some workings
y <- 2*x
mydata <- data.frame(y = y, x = x)
xyplot <- ggplot(mydata, aes(x = x, y = y)) + geom_point()
ggsave("output/xyplot.png", xyplot, width = 7, height = 6, dpi = 200)
```

4. Save the script file as "first-commit.R". Run the script file. Verify that both the script file, the data, and the output plot are saved in the project directory.

1.3.4 Commit the changes you just made

Commiting saves changes locally – like taking a snapshot of your files at a particular point in time.

1. In RStudio, click the "Git" tab.
2. Choose the files you want to commit by ticking the "Staged" box (for those you want to commit). Particularly early on, think about what you do and don't want to commit and push. Its advised to push source code that creates data and plots, rather than the data and plots themselves. Also, the way we are using GitHub, everything you push is publicly available, so you may not want to put your data or brilliant new idea up for all to see and use. Private repos are available and just cost a bit of money or additional time and effort to set up. Note though, that you can commit and push any file – data, figures, as well as code.
3. Click "Commit", which will open up a new pane. Here you'll be able to see any changes to existing files that you'll be committing. For now all the files are new, so there's nothing to see. For each commit, you need to add a "Commit message" in the text box provided. For now, type "first commit" or similar, and then click "Commit".

Your changes are now committed.

1.3.5 Push your local commits to GitHub

Pushing saves your changes on the remote repository. You'd always commit first, before pushing, so these steps start off at the end of a commit step, just after you click "Commit".

1. Click the blue "Pull" tab. This checks that there is not a more recent copy of your files on GitHub. If you're working on your own this will obviously be the case, but if you are collaborating someone may have pushed their changes since you last pulled, and this can create conflicts. Pull-before-push is therefore a useful habit to get into. You should get a message saying "Already up-to-date". Close this window.
2. Click the green "Push" tab. This will push your changes onto the GitHub repo. Once complete, you can close the window and the main pane you have been using. Go to your repo on GitHub, refresh the browser, and check that the changes have been uploaded.

1.4 Collaborating with others on GitHub

This section assumes there is an existing GitHub repo that you would like to interact with, by using the code in the repo and/or by contributing code to the repo yourself. These steps will also work in *starting* a collaborative project – you'll just start by forking an empty repo (belonging to the person you're collaborating with).

1. Fork the target repo to your own GitHub account. Do this by browsing to the target repo on GitHub and clicking the "Fork" button. This creates your own copy of the repo.
2. You can now clone the repo to your local machine. From *your* GitHub account, get the repo's URL by clicking the green "Code" button.
3. Start a new R project using version control as shown before (*File > New Project > Version Control > Git*). Enter the URL you copied in the previous step. You should see the contents of the target repo in your R project.

4. You can now work on the project and commit and push any changes in exactly the same way described previously. If you're not interested in developments in the target repo, or in collaborating by contributing changes to the target repo, then you don't need the following steps. Often though, you will want to keep your forked repo up-to-date with the target repo (either to keep track of new developments or because you are working together in some way). In that case, read on.
5. To keep a forked repo up-to-date with the target repo, you first need to configure a remote that points from your repo to the "upstream" target repo. First check the current remote repository for your fork by opening the terminal, browsing to the project directory and typing `git remote -v`. You should see the URL of your repo.
6. Add the upstream target repo by typing
   `git remote add upstream https://github.com/ORIGINAL_OWNER/ORIGINAL_REPOSITORY.git`.
   Type `git remote -v` again to check that it has been added.
7. You can now pull any updates from the target repo into your local repo (fork) with `git pull upstream master`.

If you are going to be contributing changes to the target repo, you should now create a new branch and do your work there, rather than working directly in the master branch. When you are ready to push your changes upstream to the target repo, you do this by opening a pull request:

1. Browse to the main page of the target repo.
2. In the "Branch" menu, choose the branch that contains your commits.
3. Click "New pull request".
4. Choose the branch of the target repo you want to merge your changes with.
5. Enter a title and description for your pull request.
6. Click "Create pull request".

The pull request is then sent to the owner of the target repo, who may accept the request, ask for additional changes, or reject the request.

1.4.1 Sources and further resources

1. http://happygitwithr.com/
2. http://www-cs-students.stanford.edu/~blynn/gitmagic/
3. https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting
4. https://github.com/blog/2019-how-to-undo-almost-anything-with-git
5. https://guides.github.com/activities/forking/
6. https://help.github.com/articles/fork-a-repo/
7. https://help.github.com/articles/configuring-a-remote-for-a-fork/
8. https://help.github.com/articles/syncing-a-fork/
9. https://www.youtube.com/watch?v=u-kAeG4jkMA