Week 0: Version Control 101

Scene 1: Version Control 101

Welcome to this very basic introduction to version control. We've seen how to document our code for reproducible research. Why should you go even further and put your code on a place like GitHub? This video will discuss why this might be a good idea, how it works, both in a simple scenario and if you want to try out some crazy ideas, and how to get started.

Scene 2: An Inconvenient Truth: Documents Evolve!

It's time to face an inconvenient truth in research: documents are not static, they evolve. And this applies equally to R code, data sets, and manuscripts. You may have seen this: first I have a draft, then a final version, then comes final-final, final-revised, ultimate, ultimate plus one.

Half a year later I need to do a new analysis because some reviewer requested it, and I have no clue anymore which is the latest version. I dimly recall that I had tried something like this last year, but where? Oh, and there was a bug in the data too, did I run my code with the old or the new data set?

Scene 3: Version History

If we label the versions with v1, v2, v3, etc., we know at least what chronological order they were in. Version history goes a step further and annotates each version with a brief description of the change made to the document.

Scene 4: How does it work?

How does it work? It starts with what we already do: save our work, and save it often! This should be second nature just like brushing your teeth. Version control adds three more steps to your data hygiene - think of it as the flossing part.

Step 1 involves identifying which of the files in the workspace that have been changed need to be checked in. This is called staging.

Step 2 checks the changed file into the local repository on your computer. This is called a commit.

Step 3 syncs the changes to a remote repository. This is called pushing.

We can also download the current version from the remote repository. If you actively pull the latest version from somewhere else, that is called pulling.

Scene 5: Simple Scenario

Let's consider a simple scenario, where I just want to keep track of the changes as I go in a linear fashion. This means that I have a single current copy of my project. That is called my local master. I also have a current version of it in the remote repository, that's the remote master. Right now, they are synced and thus identical.

Now I make some changes in two files to fix some errors. The files will automatically show up under 'unstaged changes'. When I check them off, they are moved to the staged changes. All I need to do is enter a commit message and hit 'commit'.

This commits the changes to the local master and annotates them with my commit message. But now the local master is one commit ahead of the remote master. I need to actively push the changes to the remote master, this does not happen automatically. Now both are at the same version again.

This goes on and on, and I may even be ahead of the remote master by a few commits, but it is a good idea to stage, commit and push often. In this simple scenario, you can think of it as your backup system.

Scene 6: Got some crazy ideas?

If I want to try out a crazy idea, I can create a branch to keep the changes separate from the master version of the project until I'm convinced that it was a good idea and I actually want to implement it.

I start again with a local master and a remote master. Now I create a new branch called 'my idea', and I push it to the remote repo. I do some programming in the new branch, and I can work on the master in parallel.

Then I create a new branch because I had a better idea, and I push it to the remote repo. I quickly abandon the original idea, the branch does not continue further, but I continue working on the new idea. I also had another, totally unrelated idea, and I create a separate branch for it. That one was a success, and I merge it with the master. Finally, the second idea turns out to be good as well and I merge it with the master.

When I merge these different versions, some parts of the code may be in conflict. I can view them side by side and decide for each chunk of code which version I want to go with.

Scene 7: User interface

Now you know enough to intuitively understand the graphical user interface of GitKraken! On the left, we have a list of the repositories and branches, in the center, we have the commit history, and on the right we have the staging area and tools for inspecting merge conflicts.

RStudio has a less graphical interface. Once you set up your R project with Version Control using Git, you get a new tab called 'Git'. Any changed files will automatically be listed and you can stage them with a click. Above the staging area, you have buttons for commit, push, and pull, also a button 'Diff' on the left to view differences between versions. On the right, you can create a new branch, and you can switch branches in the local repository with a drop-down menu.

Scene 8: Getting Started

Once I have activated Version Control with GitHub for my project in RStudio and connected it with GitHub, the local repository on my Windows laptop is handled behind the scenes, and I can push to the remote repo in the cloud with a button from within R Studio, as we have just seen.

However, for setting this up, and for trouble shooting, I have to use the Terminal or shell. I don't really like that, and I do prefer a more graphical interface. That's why I have installed GitKraken. There are other alternatives like Git for Windows, which actually runs on Macs as well. I use GitKraken to manage repositories and anything related to merging branches.

Once I have pushed my current version to GitHub in the cloud, I can pull it from my Mac at home. I use the same software combination there, with RStudio and GitKraken. And after working on my Mac, I push to GitHub and pull from there to the laptop. In a similar way, you would collaborate with others, but we'll look at collaboration in a later video.

I have a warning, though. Usually I sync all my files between my machines with Dropbox. That's not a good idea for my R projects with version control, and I keep them on the Desktop outside of Dropbox. The reason is that the code files are saved with Windows encoding on my laptop and with Unix encoding on my Mac. That means that the line endings are coded as 'CRLF' on one side and as 'LF' on the other side. Github takes care of this, but Dropbox doesn't.

This brings us to the end of Version Control 101. Check the 'Week 0 Worked Example' for further instructions how to install and work with version control in RStudio.