L1: Replicability and Version Control

James Okun

MIT

January 23, 2024

Today We Will Learn...

- How to organize your data and code in a way that is compatible with most replication guidelines
- How to use Git for version control

Best Practices in Coding and Data Management

Common Problems in Empirical Research

This section is based on Gentzkow and Shapiro (2014)

Do these panic situations from Gentzkow and Shapiro (2014) ring a bell?

1. "In trying to replicate the estimates from an early draft of a paper, we discover that the code that produced the estimates no longer works because it calls files that have since been moved. When we finally track down the files and get the code running, the results are different from earlier ones"
2. "A referee suggests changing our sample definition. The code that defines the sample has been copied and pasted throughout our project directory, and making the change requires updating dozens of files. In doing this, we realize that we were actually using different definitions in different places, so some of our results are based on inconsistent samples"
3. "We and our two research assistants all write code that refers to a common set of data files stored on a shared drive. Our work is constantly interrupted because changes one of us makes to the data files causes the others' code to break"

Examples from the AEA guidelines

This slide is based on the AEA replication guidelines

Some quotes:

1. "Every replication package requires a document outlining where the data comes from, what data is provided, what requirements are needed to run the code in the replication package, how to run the code, what results to expect, and where to find the results"
2. "The replication package should reproduce the tables and figures, as well as any in-text numbers, by **running code without manual intervention**"
3. "A master script is strongly encouraged"
4. "You can assume that replicator can manipulate a top-level configuration file, for instance, to set a base directory, but not setting a base directory at the top of 25 different files"
5. "You should NOT assume that the replicator has any of your packages/modules/etc. installed: provide a setup program to install these (not manual instructions)"

Best Practices

Gentzkow and Shapiro (2014) suggest a series of simple rules that can make your life easier in that respect if you implement them thoroughly from the beginning of your project

1. **Automation**
2. **Directories**
3. **Keys**
4. **Abstraction**
5. **Documentation**

Best Practices

(1) Automation

- Automate everything that can be automated
  - Write scripts (e.g. `.jl` or `.do` files) rather than using the command line
  - Your scripts should take care of loading the data and saving the outputs, rather than doing it manually
- Write a single shell script that executes all code from beginning to end
  - This is important for replicability: it makes sure that all files are run in the correct order
  - Include the data cleaning, the analysis and if possible also compiling the paper and slides from latex (tables and images can be automatically updated this way)

Best Practices

(2) Directories

- GS recommendation: separate directories by function
  - These can be the steps in the analysis, for example: `/build`, `/rf`, `/structural`, `/paper`, `/slides`
  - Each one can have `/input`, `/code`, `/output` and, if needed, `/temp` sub-folders
  - Each one can have its own shell script that runs just that particular step
  - Sometimes the `/input` subfolder will contain a link to the `/output` subfolder of the previous step
- The raw data should be on its own directory, and ideally write-protected (so that you don't accidentally modify it in an irreversible way!)
- Use pointers: e.g. create a variable "data_dir" (a global on Stata) that points to the data folder. This way, if you change the location of that folder, you only need to modify one instance.

Best Practices

(3) Keys

- Keep cleaned data organized in tables with unique, non-missing keys
  - A key is a set of variables that uniquely identify an observation in a dataset. This could be a single id (for cross-sectional data or time series), or multiple ids (for panel-like structures)
- Sometimes it can be a good idea to keep data normalized as far into your code pipeline as you can:
  - By *normalized*, GS mean that each data file contains attributes specific to the units represented by its key:

    "County population is a property of a county, so it lives in the county table. State population is a property of a state, so it cannot live in the county table. If we had panel data on counties, we would need separate tables for things that vary at the county level (like state) and things that vary at the county-year level (like population)"
  - They also recommend doing all sorts of manipulations (e.g. computing the log of population) at the normalized stage, before merging the data files for analysis

Best Practices

(3) Keys (example)

Figure: Example of normalized datasets

county	state	population
36037	NY	3817735
36038	NY	422999
36039	NY	324920
36040	NY	143432
37001	VA	3228290
37002	VA	449499
37003	VA	383888
37004	VA	483829

state	population	region
NY	43320903	1
VA	7173000	3

Best Practices

(4) Abstraction

- Abstract to eliminate redundancy
  - If you need to perform an operation several times (e.g. computing a leave-one-out mean of variables), it is better to define a function that does it
  - It avoids having to copy-paste the code that computes the leave-one-out mean every time (with all the opportunities for error that that creates)
  - Functions can also make your code more readable and clear
  - Test your functions on fake data or small subsets of real data to verify that they work as intended
  - Bonus: in Julia, abstracting operations into functions will greatly reduce runtime, as we saw in the previous lecture

Best Practices

(5) Documentation

- Write readme files and comment your code to help your future self and others understand your code
  - Keep your documentation updated: if you change some aspect of your code, think whether the comments should also be updated. In general, avoid mentioning results / outputs in the comments as those are subject to change
- Code should be self-documenting
  - At the same time, if you write code that is clear enough, it will be clear what each piece is doing most of the time without need to comment
  - Julia is one of the most readable languages in that sense

Additional Resources

Courtesy of MIT Libraries

- CarpentriesMIT is a group trying to build a local learning community for reproducible research and data skills. They organize occasional workshops
- MIT Libraries teach a series of workshops on data management. You can find more information, including resources from previous workshops, here

Version Control Using Git

This section is based on guidelines from SEII and Blueprint Labs

- Sometimes we want to keep track of changes in our code. One way to do it is to save different versions of the code, as: `Analysis_v1.jl`, `Analysis_v2.jl`, `Analysis_final.jl`, `Analysis_final_final.jl`, `Analysis_THE_FINALEST.jl`
- Instead, consider using version control software, like Git
- Git allows you to track changes of text-based files over time and to access old versions. It improves your ability to work collaboratively, find and fix bugs and replicate projects
- By default, Git operates out of the command line. GitHub is a cloud based service that stores Git repositories on the cloud and allows users to make copies locally. It has a nice GUI called GitHub desktop that makes it even easier to use

- A Git repository (repo) is a project folder that contains the code and other files that we want to track
- Repos can either be **bare**, if they contain the history of changes but not a physical copy of the files, or **non-bare**, if they contain the history of changes and a physical copy of the files
- A **remote** repo is a bare or non-bare repository that is hosted in a location accessible to all users. It holds a common version of code history that everyone can see and access
- A **local** repo is a non-bare repository in your personal directory. This is where you can develop code, and then send your work to the remote repo so that other team members can access it

- Git works by taking snapshots of your code at different points in time. Each of these snapshots is called a **commit**. You decide what changes to place onto a commit; you can even save all your most recent changes while only selecting some to commit
- Some best practices:
  1. Better to overcommit than undercommit (more tracking points), but there is a trade-off between quantity and quality
  2. Divide different types of changes into different commits (even within the same file)
  3. Accompany commits with meaningful messages, to help your future self or colleagues understand what each commit did

- **Pushing** sends the changes you committed in your local repo into the remote repo
- **Fetching** does the reverse: it updates the local repo with any changes made at the remote
- **Pulling** is a fetch + a merge: in addition to retrieving changes, you will merge the updated remote branch into your local version of the branch (more on branches to follow)
- **Stashing** stores a copy of all your uncommitted changes to files and reverts the files back to their unedited state. This can be helpful if you need to switch branches temporarily but aren't ready to commit your changes yet (since you can't switch branches with uncommitted changes)

- In general, you only want to track changes in your code files, although your local repo folder will contain other files.
- One example of things you might want to ignore is data (for privacy/legal reasons or because they are very large files). Instead, make sure that all your collaborators have access to the raw data and track the code files that produce the clean dataset
- Another example is outputs: graphs, tables, pdf-ed LaTeX. PDFs, XLS and other non-ASCII files are not suitable for git. Just track the code that produces them instead, and you should be able to recover them from there
- The ignored files are declared in the `.gitignore` file. I'll show you how to access it in the example at the end

- Github Issues are a very useful tool to keep track of your to-do list
- There are several ways to create issues and to mark them as resolved after they are addressed. The easiest is probably from the Issues tab on Github
- You can organize and prioritize issues with projects. To track issues as part of a larger issue, you can use task lists
- You can use #labels to categorize issues, and @mention your collaborators so that they get a notification about issues

- **Branches** have two primary uses in terms of workflow:
  1. You can work on distinct types of code edits separately by using multiple branches
  2. You can keep a **master** version of the code that runs top to bottom without issues, while having a **develop** branch where you work on new code
- We will talk about A Successful Git Branching Model
- You can incorporate changes made on one branch into another branch by **merging**

- A merge conflict arises when there are changes made to the same line of a code file on the two branches you're trying to merge. For example, my STATA code has the line `reg Y X`. I create a branch "robust" where I try `reg Y X, r` instead, an another branch "controls" where I try `reg Y X W` instead. If I try to merge "robust" and "controls," I will have a merge conflicts
- This doesn't happen if the discrepancy corresponds to different files or even different lines on the same file: the merge keeps the most recent version of those
- Merge conflicts need to be resolved. To do that, open the problematic file, handpick the changes that you want to keep, delete the rest, save the file and commit (this can be done on GitHub)

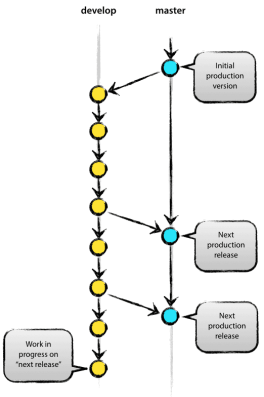
- Some best practices to avoid merge conflicts:
  1. In your local repo, avoid working on the same section of code on two separate branches if you intend to merge them together in the future
  2. Avoid working in the same file as others on the project
  3. If you realize you've both made changes to the same file, stash your changes, pull their changes, apply your original stash, make any necessary adjustments, and push back to the origin

A successful Git branching model

The main branches

This section is based on a blog post, *A successful Git branching model* by Vincent Driessen

- The repository will be organized around two main branches with an infinite lifetime: **master** and **develop**
  - The **master** branch should hold stable versions of the code that are ready to be run
  - The **develop** branch is where those stable versions are built, and it can contain work in progress at any point in time
  - Once your **develop** branch reaches a stable point, all changes should be merged back into **master**. It is recommended to tag the merge with a version number



A successful Git branching model

The main branches

Here is how to merge **develop** into **master** from the command line:

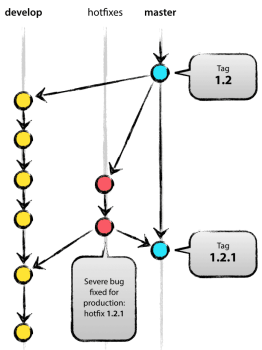
```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff develop
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2
```

The `--no-ff` tag causes the merge to create a new commit object so that all the commits in the merged branch before the merge are grouped together (so they can all be reverted at the same time). See the blog post for more info on this

A successful Git branching model

Supporting branches

- Supporting branches have a specific purpose and a limited life time. These can include:
  1. **feature** branches. Imagine you want to try something (e.g. a different measure for the dependent variable). You can try that in a feature branch created for that purpose. These branch off from **develop** and, if you're happy with it, must merge back into **develop**, or otherwise, be discarded.
  2. **hotfix** branches. Imagine you discover a bug in the **master** code! This needs to be fixed ASAP. You can create a specific branch for that, and merge it back into *both* **master** and **develop**.



A successful Git branching model

Supporting branches

Here is how to create a **hotfix** branch, merge to **master** and **develop**, and delete it from the command line:

```
$ git checkout -b hotfix-1.2.1 master
Switched to a new branch "hotfix-1.2.1"
[...]
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2.1
```

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was abbe5
```

A successful Git branching model

Collaborating

- Sometimes you will want to work in parallel with other people
- In that case, it may be a good idea to have a central version of the repo, called the **origin**, from which every team member pulls and pushes, while every developer can have their own **develop** branch locally
- One team member may want to pull changes from another team member's repo. This can be done with a Git remote that points to that repo