Notes on Project Management and Collaboration

Matt Wibbenmeyer

This document provides some notes on the standard workflow and file structures I use for my projects and collaborations, including work with RFF RAs and interns. There are three good reasons to have a standardized method of project organization and workflow. First, having a standard system of file organization makes it easy to jump across projects, and locate relevant files, including scripts, data sets, and figures. Second, and relatedly, good organization facilitates reproduceability. This is important as a scientific principle (it is important that other researchers can inspect and reproduce findings), but it is also important that you can reproduce *your own* work. Projects frequently last years, so it is important to have an organized record of what you've done and why you did it. With a project's scripts as well as its raw data, it should be possible for another researcher (or future you) to entirely reproduce the results of the project. Good organization helps facilitate this. Finally, good organization helps faciltate collaboration. To that end, the system of file structures described here plays well with Git and GitHub, which is useful for sharing code and file structures among researchers and across machines.

In the first section, this note discusses the primary folders I use to organize a project. In the second section, I discuss file organization and naming conventions. In the third, I discuss basic use of Git and GitHub and provide links to further resources.

1. File Structures

A *project* is collection of data, scripts, tables and figures, and text, which will ultimately—if all goes well—get published in an academic journal or other publication. The data is the raw material of the project, and is usually collected from some source. Scripts are written in a computer language (perhaps R or Stata) to manipulate and analyze the data. Finally, the paper summarizes the goals, methods and results of the project. Each of these constituent components of the project is stored in the *project folder*, which we will call `project/`.

The subdirectories within the project folder store the various constituent parts of the project. These subdirectories are `raw_data/`, `scripts/`, `processing/`, `results/`, and `paper/`. The purposes of each directory are described in the following subsections.

1.1 Raw data

The `raw_data/` folder stores untouched, unmanipulated raw data. This data may have been downloaded from an online source, or it may have been obtained via the mail on a CD-ROM or USB drive, or it may have been entered by hand based on tables from `.pdf` documents. It may consist of `.csv` or `.xls` files, or it may consist of geospatial vector or raster data. The important thing is that the data in this folder represents the unprocessed, unadulterated raw material of the project. Care should be taken not to alter the contents of files stored in the raw data folder (for example, by opening an `.xls` file and changing the contents of a cell). As well, it is best practice to include `readme.txt` files within appropriate subdirectories of the raw data folder to indicate where the data was downloaded or otherwise obtained.

1.2 Scripts

The `scripts/` folder contains all of the scripts (i.e. code) that are used to manipulate and analyze raw data, or data that has been processed by other scripts. Therefore, the folder may contain a mix of `.do` (Stata), `.R` and `.Rmd` (R), and `.py` (Python) files, among others. Because the scripts provide all of the details regarding the manipulation and analysis of the raw data, it should be possible to entirely reproduce the results of the project based on the contents of the `raw_data/` and `scripts/` folders.

If possible, the scripts folder can also include scripts to download the appropriate raw data. For example, if the project uses Census data, the scripts folder can include scripts using the R library `tidycensus` to download the relevant data. If the data is downloaded from a static web link, a script can be written to download the data set using the `download.file()` function in R, for example.

1.3 Processing

This folder contains any intermediate data products that have been produced from the raw data, or from other intermediate data products, using scripts. Data sets stored in the `processing/` folder may be `.csv`, `.xls`, `.dta`, or `.shp` files, among various other file formats. Any scripts that manipulate the raw data in any way—data cleaning, merging, creating new variables from existing variables, etc.—should save output data sets to the `processing/` folder.

1.4 Results

Like the `processing/` folder, the contents of the `results/` are also produced from the raw data or other intermediate other data products using scripts. The folder contains all tables and figures produced for the project. As well, it may contain files like Stata `.est` files, which store estimates produced by regressions in Stata. Results stored in this folder may include tables and figures used for exploratory analysis, as well as tables and figures produced for publication.

1.5 Paper

The `paper/` folder contains written material relevant to the project. Most importantly, it contains the `.doc` or `.tex` files associated with the paper being produced for publication. The folder also contains presentations or notes related to the project.

2. Organization and naming conventions

A file's path and name should be informative about the file's purpose and role in the project; however, files can be organized and named so that more information is contained in either the file path or file name. For example, suppose we have collected raw data on crop prices across US counties for various crops and years, which are being used to construct per acre net returns for cropland by count. The data for wheat prices in 2007 could be stored as `raw_data/wheat_prices_2007.csv`. The problem with this method of file naming is that the `raw_data/` folder will become crowded with files, especially if we have many crops and years and *especially* if the project includes raw data other than just crop prices. Alternatively, the data could be saved as `raw_data/net_returns/crops/prices/ wheat/wheat_prices_2007.csv`. Saving the data this way will result in a file structure that is organized and descriptive regarding the purpose of the various data sets used in the project.

In the case of scripts, a second naming convention applies; scripts (and the folders within which they are contained!) should be numbered so that a user can easily see in what order the scripts should be run. For example, suppose a project consists of several steps: cleaning and merging the data, providing descriptive statistics, and running regression analyses. The scripts directory may then contain three subdirectories:

- `scripts/01_data/`
- `scripts/02_descriptives/`
- `scripts/03_analysis/`.

Within the data folder, data cleaning and merging may include several steps, for example: cleaning the outcome variable data set, cleaning the predictor variable data set, and merging the two data sets. Within the data folder, the scripts can be named:

- `scripts/01_data/01_clean-outcome-variable.R`
- `scripts/01_data/02_clean-predictor-variable.R`
- `scripts/01_data/03_merge-data.R`

A side-benefit of numbering the files in this way is that folders can be sorted in Windows Explorer (or the Finder on Macs) so that scripts are listed in the order they are to be run.

3. Collaboration using GitHub

Git and GitHub provide a platform for collaboration and version control. You can think of GitHub as providing a Dropbox-like platform for code, which incorporates a sophisticated version of Track Changes. While Git provides many features and commands, basic use is very simple. This section will discuss basic use of GitHub. Many resources for more advanced use of Git are available online, including these excellent notes by Grant McDermott.

3.1 Cloning a repository

When you first start work on an existing project, you will want to “clone the repo” to your local machine. This means that you will download the project and its contents (as well as a hidden `.git` directory, which specifies the connection between your local folder and the online repository) from GitHub.com. The steps to clone the repository *`repo-name`* owned by user *`username`* are as follows:

1. Confirm with the owner of the repository that you are an approved collaborator.
2. In the command prompt (or Terminal on Macs), navigate to the directory you would like your project to be stored in. If you are unfamiliar with using the command prompt or Terminal, Grant McDermott also has excellent notes on this.
3. Enter the command `git clone https://github.com/`*`username`*`/`*`repo-name`*`.git`.
4. Enter your GitHub username and password.

The project will then download to your chosen local directory.

3.2 Making changes to the project

When you make a change to the project (for example, by creating a new file or by editing an existing file), you will want to save that change and upload it to the repository so that other collaborators can view it.

1. **Staging changes.** The first step to saving (“committing”) and uploading (“pushing”) changes is to add your changes to the staging area. Suppose you added a new script to the `scripts/` folder, and made a change to a second script. You want to save (“commit”) those changes, so from the project folder in the command prompt window (or Terminal), you would type:

   ```
   git add scripts/*
   ```

   This will add all of the changes that have been made within the `scripts/` folder since the last commit to the staging area. If you want to add any changes to the entire project since the last commit to the staging area, you could type:

   ```
   git add .
   ```

   In general, it’s better practice to commit fewer changes at a time, and to commit related changes with one another. This facilitates providing more descriptive commit messages (described below), and better facilitates reverting to appropriate previous commits when that is necessary.

2. **Check status.** It is a good idea to check the git status after you have added changes to the staging area. This provides an opportunity to ensure that no mistaken files have accidentally been added to the staging area. If a very large file (¿100 MB) is mistakenly added to the staging area and gets commited, the commit will need to be removed before any further changes can be pushed, and this can sometimes be very messy depending on how many further commits have been made. To check status, type:

   ```
   git status
   ```

3. **Committing changes.** Once you have staged the appropriate files and changes, they can be committed. In the language of Git, to "commit" is to save an updated version of the project that contains the changes you have added to the staging area. To commit, type:

   ```
   git commit -m ‘‘My first commit!’’
   ```

   While "My first commit!" is an okay celebratory first commit message, future commits should provide a short description of the changes that have been made in that commit. If there is ever a need to revert back to a past commit, this makes it easier to find the right place to revert to.

4. **Pulling.** Before you "push" (upload) your committed changes to the online repository hosted on GitHub.com, it is a good idea to "pull" (download) any changes that may have been made by other collaborators since you last pulled. To pull, type:

   ```
   git pull origin master
   ```

   A problem can arise if a collaborator has made changes to a file that you also have been editing. In this case, Git may flag the file as having conflicts. If this happens, you will need to open the file, manually edit it to resolve conflicts, add the file to the staging area, and commit the changes.

5. **Pushing.** Finally, you can push your changes to the online repository on GitHub.com. To push, type:

   ```
   git push origin master
   ```

   Now, the files can be viewed online by your collaborators, or pulled so that they are available on their local machines.

3.3 The `.gitignore`

The `.gitignore` file is an important hidden file contained within a project's root directory, which specifies which files should be "tracked" by Git for committing and pushing to the online GitHub repository. For example, suppose you have made changes to a data processing script stored in `scripts/`. You have run the script, and saved the output within the `processing/` folder. If the `processing/` folder is set to be ignored by the `.gitignore` file, but the `scripts/` folder is not, when you type the command:

```
git add .
```

your changes in the `scripts/` folder will be added to the staging area, but your changes in the `processing/` folder will not.

The `.gitignore` is useful because it forces Git to routinely ignore large files. GitHub has a file size limit of 100MB; no files larger than 100MB can be pushed to an online repository. Script files will not exceed the maximum file size limit, and neither should `.tex` files, `.pdfs`, and most image files. However, data sets will frequently exceed 100MB in size. To avoid committing large files, we can store them in specified directories that are set to be ignored from tracking by the `.gitignore`.

This explains the project folder structure described in Section 1. Folders containing files we want to be tracked and uploaded to the GitHub repo (scripts, `.tex` files, tables and figures) are kept separate from folders containing data sets, which would be too large to push. The `scripts/`, `paper/`, and `results/` folders are tracked, whereas the `raw_data/` and `processing/` folders are ignored.

An example `.gitignore` file is printed below:

```
/*
/*/
!/scripts/
!/results/
!/paper/
```

This tells Git to remove from tracking all folders and their contents *except* for the `scripts/`, `paper/`, and `results/` folders.

Because the `raw_data/` folder is not shared through GitHub, its contents must be shared another way. There are several possibilities here. My preferred method is to keep a 'master' `raw_data/` folder in the project folder on the RFF `L:/` drive. This way, any project collaborators can log on to the server and access not only the raw data sets used in the project, but the file directory at which they are stored. The file path for the raw data can then be replicated on the users' local machine after they download it from the server.

An alternative is to use Dropbox, or another file-sharing service, to share raw data files. In this case, the appropriate file paths for raw data files can be obtained from the scripts that reference them. This can work, but it tends to be less systematic. There is also a possibility on large projects that no individual user will have the *entire* `raw_data/` folder, which creates a risk that data will be lost.