ESOC Research Production Guide, V1.0

February 2019

Jacob N. Shapiro[1]

1. Summary

The following guide is intended to provide a common set of research practices for the Empirical Studies of Conflict group at Princeton, as well as colleagues elsewhere. Practically speaking, following this guide should ensure that (a) team projects can be managed with minimum friction and (b) any research affiliate will be able to get up-to-speed on any given project rapidly and efficiently. The conventions detailed below should allow new colleagues to begin work and collaboration quickly.

This guide was motivated by repeated experiences where research projects without clearly articulated common standards of communication and deliverable production became extremely inefficient and hard to revisit/reproduce years later. This document is not an exhaustive template, but it organizes effective practices from various projects. Keeping these practices in mind will reduce researchers' long-term workload.

2. Motivation and Acknowledgements

Common standards are particularly useful to manage the otherwise unwieldy scale of projects that involve large teams or multi-year efforts, and enable the same or completely new researchers to re-use work at later dates. The reproducible research practices describe here also enable others to understand analyses end-to-end by examining a single shared zip-file, in addition to reproducing a project's exact results.

The practices outlined within this document come from a variety of sources, first and foremost, feedback from ESOC researchers and colleagues. Other sources include an adaptation of Professor Scott Long's 2017 "Lecture and Lab Notes for Strategies for Reproducible Research," Christopher Gandrud's 2014 book, Reproducible Research with R and RStudio, Gentzkow & Shapiro's 2014 "Code and Data for the Social Sciences," Miriam Golden's "Guidelines for Authorship, Awarding Credit, and Intellectual Property Rights", and the Evidence in Governance and Politics (EGAP) guidelines on pre-analysis plans.

3. Documentation

[1] Ben Crisman, Dustin Dienhart, and Manu Singh provided phenomenal help in drafting this manual as did colleagues too numerous to mention. Any mistakes or bad practices advocated herein are the author's fault.

Effective documentation enables:

A. New parties to enter a project understanding its core components.
B. Yourself to return years later and recall exact reasons for your actions, and
C. Accumulation of knowledge on a long-run basis, according to the project design.

Complete ***project plans*** should include

Objectives | *Requirements* | *Data Resources* | *Relevant Literature* | *Schedule* | *Analytic Methods*

These should be detailed early on, not just within email chains. Describe plans using easily-shared documents like Google Docs and Google Sheets.

In addition to a project plan, researchers should also design a formal ***pre-analysis plan***.

This plan should:

A. Memorialize design decisions;
B. Constrain fishing; and
C. Force statistical considerations: power calculations, sampling, identification strategy, and the robustness of findings.

An example of a pre-analysis plan for and RCT can be found at the Open Science Foundation: https://osf.io/dhf25/.

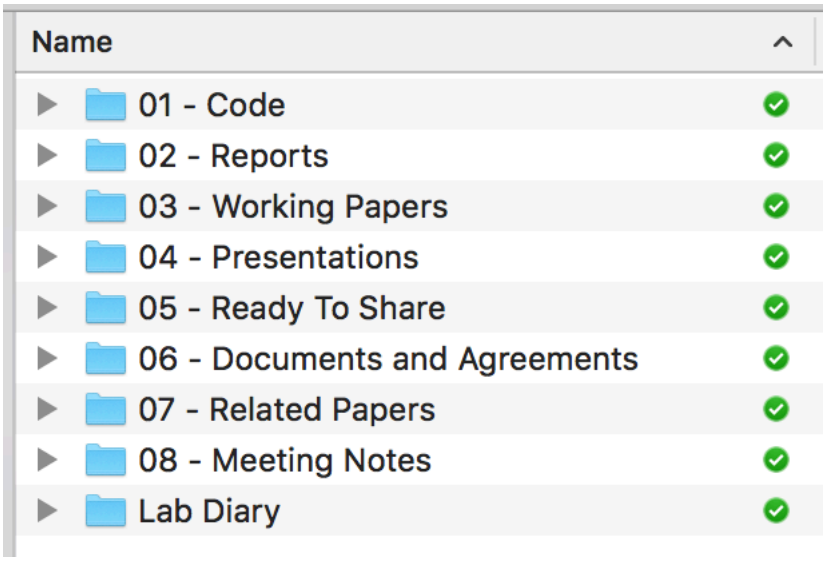
Finally, in addition to documenting the scaffolding for a project's execution and analysis, document daily project activity inside a ***lab diary***, a concise .txt file for each project. Also record all reasons for any changes to strategy or workflow. Lab diaries are uncommon in social science, but are an essential standard in laboratory sciences.

In writing these documents and the lab diary in particular, always consider the next person to join the project and how your writing can ease their startup efforts.

4. Storage and Organization

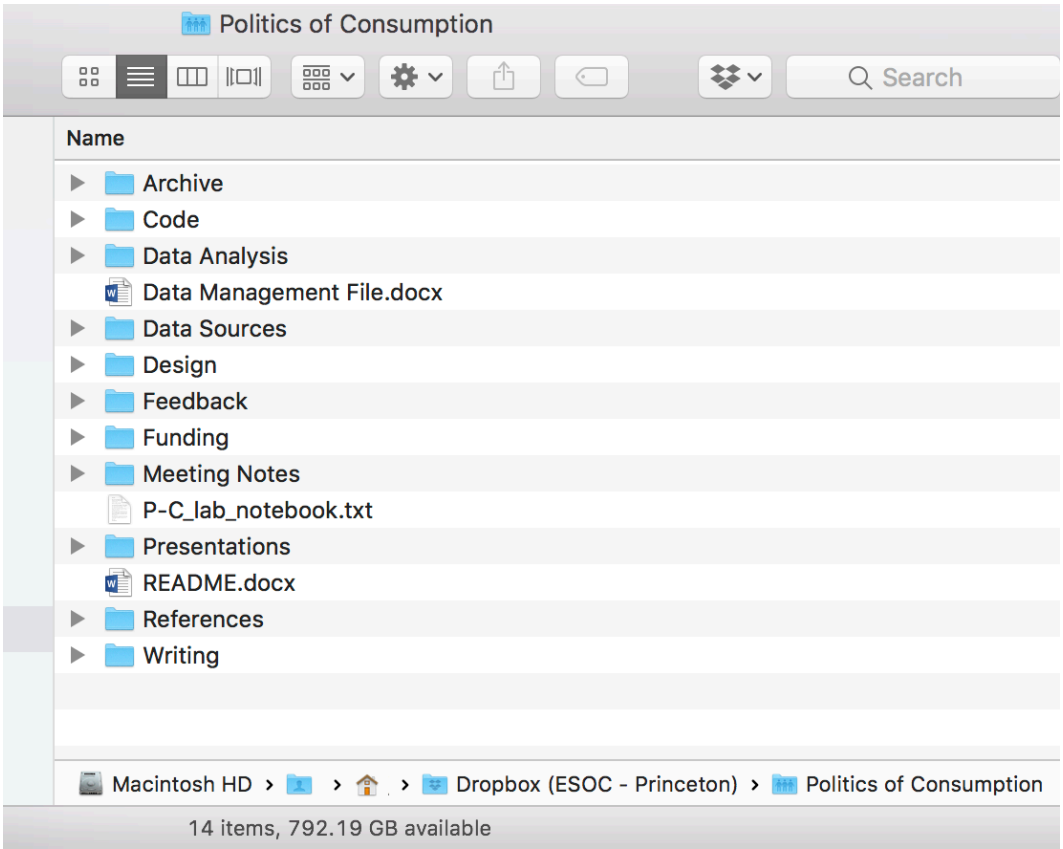
Large projects should be run entirely on shared server space, for example on Dropbox. Create a directory structure appropriate to the project. This should vary between observational studies, randomized-controlled trials, and theory papers. Ideally all work, including scratch code files and drafts of internal reports should be done in the shared space. The ideal is that anyone on the project could be hit by a bus tomorrow with minimal disruption.

There are many potential approaches to organization. One appropriate to projects that do not have a heavy data management workload is below, which contains basic necessity folders you will maintain, and also bookends the most commonly accessed folders, 'Code' and 'Lab Diary', when arranged by name. (Note: whenever beginning files or folders with numbers, pad single numbers with 0s, i.e., '01', '02'…etc, otherwise your ordering will go 1, 10, 11, 12.):



Other folders or subfolders could include *Design, Funding, References, Surveys,* and *Training Materials*.

For more data-intensive projects a directory structure which keeps data, data analysis and code separate at the highest levels is optimal. See below:



Above all, keep structures functional and simple. And document them in a readme file.

Sidenote: Do not allow ‘orphan’ or free-floating, uncategorized files. Everything should go inside a primary folder or subfolder until the final level.

When naming files, note that using version numbers e.g. ‘V3’ and dates e.g. ‘4-8-15’ can quickly become unreliable and ambiguous. The ‘Date Modified’ attribute within the operating system itself will always be reliable, and thus any user may find themselves puzzled when the most recently edited file is the revision numbered two versions or two days lower than the newest labelling-choice. Whenever possible, name file revisions based on function to avoid confusion and make it even easier for others to follow.

Keep the following in mind:

- If you do use dates, use internationally-consistent ISO 8601: YYYY-MM-DD
- Use consistent capitalization choices, based on the original author’s construction.
- Files and folders should be searchable 2 – 5 years later by machines and people. Think about choosing a name that will makes sense even 5 years later.
- Don’t rely on directory names/structure (if possible) to provide critical information about the contents of the folder. If these files are migrated elsewhere, they may lose directory structure and context. Down stream searches are also facilitated by more descriptive file names.
- Spaces and hyphens are ok for display purposes within category folders not called by code, but inside the *Code* subfolders leave no spaces or special characters in names as an extra precaution to avoid any potential string manipulation issues.
- For larger codebases, use version control software instead of sequential filenames. Better yet, use Dropbox with a git repository configured and connected to RStudio.

Most importantly:

- **Think about how to name files before saving them. File names should be transparent and/or documented in a folder readme.**
- **Do not rely on local files under any circumstance. This leads to lost projects.**
- **Everything produced must be stored inside the Dropbox or other shared server space, including prototyped code.**

5. Coding

When writing scripts, *document far beyond convenience, and ideally to the point of pain*. Every non-obvious step and most obvious ones should include concise descriptions. Self-monitor what documentation works best for your own comprehension 1 day from writing, 3 days out, a week out, and so forth. Develop the skill of noticing what attributes drop off your own comprehension when picking writing back up, so future scripters can inherit your legacy code without distress and rage. File headers should, for example, include input data, output data, and the date of

changes, and file paths should be relative so anyone else can set the appropriate top-level directory (e.g. their Dropbox) and then run the code.

Within the *Code* folder, use internal directories to separate phases: *Build* for aggregating all data into its proper formats prior to running statistical models, and *Analysis* for statistical processing and plot creation.

Within the *Code* folder, your *Data* subdirectory must be split into *Raw* and *Clean*. *Raw* should contain the complete set of original, unaltered data. *Clean* should include data as transformed for further statistical processing. This distinguishes between all data that existed as you found it, and your own methods. There will often be several steps to cleaning. You can include sub-folders for different stages as needed.

If you generate dataframes that are

A. Comprised of heavy edits/changes to your raw data files,
B. Come in many different levels of aggregation or formats (e.g. a frame by ID, one by ID-year, different geographic levels), and
C. Combination processed data generated from multiple different raw files (e.g. aggregate measures from survey data combined with aggregations of incident data),

then consider also making a *Generated* folder, containing these different cases of your data. This folder in effect also provides a means to save your intermittent progress for different cases, which is particularly helpful when munging and transformation requires hundreds of lines of code that is better broken across multiple scripts.

Writing good code requires some thought. All scripts should be modular, i.e. small and according to function, to enable rapid debugging. Avoid reliance on experimental scratch files. If you create them, quickly transition to production. Avoid overwriting the same file as you go and use loops for repetitive tasks. Separate data preparation from production of graphics and tables. If you need a new constructed variable, add it in the data preparation file. And if you are working with very large data files it will be faster if you write your code on a sample, then scale up. Run on the server if parallelizing data processing can help.

Finally, within each *Code* subfolder there can be a 'make' file configured to execute all other script sources within the subfolder, and write console output to a .txt "log file" displaying operations. (Note: 'make' file written in R or Python; not a C Makefile)

5.1 On R and Python

Why script? Using only the interactive, GUI-based sides of software suites like Excel and ArcGIS leave room for user error without recording every sequential step, and cannot process a series of statistical or geospatial calculations in rapid succession without scripting. Stata is a middle-ground between R and Python for its lack of plotting flexibility and smaller overall toolkit, but satisfies our baseline priority: scripting analytical processes.

Given the choice, an onboarding researcher should prioritize tools as follows:

Python = R > Stata > Arc ~ QGIS. Never manipulate data in Excel.

If the researcher has already come in with Excel programs (unlikely) or Arc/QGIS-based Python scripts, these can be used and/or reworked into raw R / Python for maximum reproducibility. One example of a good use case would be Arc/Q's toolboxes for animated time series maps.

Wherever possible, however, use R and Python. And, generate markdown in R/Python.

5.2 Preliminary Analysis/Reports vs. Final Documents

Both R and Python have excellent infrastructure for creating notebook- or article-like markdown that clearly and effectively communicates ideas and methods, RMarkdown and Jupyter Notebooks. These can be created in native R and Python, erasing the need to pour time into TeXStudio and the like at the initial analysis stage. Within RStudio, for example, one can create an RMarkdown document and export it to a PDF automatically and immediately. (Stata also has this ability through the *texdoc* package and knowledge of various table output commands to .tex, but the R and Python resources on markdown generation are higher quality.) References for this process can be found on rstudio.com through "cheatsheets," and throughout StackExchange.

Rather than manage multiple scattered tools and environments, a single .Rmd document can source all scripts, generate relevant files for analysis, and export both a full PDF report and image files in one fell swoop. Each coded portion allows editors to decide whether to display code segments, output, and customizable annotations/descriptions. Similar functionality exists within RStudio for interactive applications (Shiny) and PDF presentations (Beamer). If you do analyses in a particular language, especially R or Python, it is worth learning how to automatically generate full deliverables using that same language. One can generate, annotate, and format an entire article using with RMarkdown from within RStudio itself and there are templates for several journal formats available from packages such as *rticles*. Documentation on the best ways to generate reports are easy to find (search "rmarkdown academic article"). A

Tools like RMarkdown are extremely useful for intermediate deliverables, but most academics, write articles in LaTeX and/or something like Overleaf. If that's the case for a given project you should have "Figures" and "Tables" subfolders in the "Writing" directory for the article/presentation. Your code should export images and tables directly to that. There are many options for doing so in Python, R, and Stata. Coordinate with your co-authors/PI on which one to use and stick with it throughout the paper.

5.3 On Programming with Clarity

When writing scripts, make code human-readable such that future researchers can follow and learn, and reduce both your own and future human error by employing functions and loops

instead of repeating code. Every copy-paste is a chance to reuse the wrong variable, perhaps containing the wrong manipulation of the data leading to false conclusions.

If project code uses uncommon packages or methods, either

A. Add a few lines to your code which detects whether the packages are installed at the beginning of the script, and installs + loads them if they're not, or
B. Load them into the Dropbox itself (for researchers to drag into where their other R packages are stored locally).
C. It's always easiest and more helpful downstream to disaggregate columns that have long textual data. For example – with addresses – never store the whole address as a string, but break it down in building, neighborhood, city etc into separate columns.
D. Even though it seems redundant - flat files are almost always better for analysis than relational databases.
E. Whenever creating a panel dataset - spend a couple minutes calculating the dimensions of the proposed output dataset. Ideally create a blank data frame of said dimensions and then fill in the required values. When debugging code check that the dimensions of your data are what you expect at every step. This simple data integrity check can save you hours and hours of pain.
F. The same advice is applicable when merging datasets. Estimate the size of the output dataset and make sure the new merged dataset is EXACTLY that dimension. Researchers could inversely mix up between left joins, right joins, cross joins and outer joins all of which lead to output datasets of different dimensions. So it's best to know what is intended before coding.
G. If you have input datasets that are in another language, it's always helpful to include a pre-processing step which translates all values, column headers, and any variable labels into English.

In addition, source any links to relevant literature or documentation you found helpful. This includes Stack Overflow threads you used to write the scripts. Make every attempt to save future researchers weeks of time and web queries.

6. Data Preparation

As stated before, we store raw data in a folder containing only *incoming* data, e.g. *Project* > *Data* > *Raw* (or similar). Within this folder, add a .txt file describing the origin of all data, even if obvious by the filename and contents, and include any related documentation or codebooks. At your discretion, include documentation from the research community (labelling project and origin). When using data from public sources, record the version number, download date, and include any relevant code that makes an exact API call required for the data pull (archive a copy of this data periodically if possible).

7. Managing Surveys and Survey Data

Managing survey production is hard. Some key points:

a. Separate out folders for documents related to design, those related to sampling, drafts, and final versions.
b. If a survey is happening in multiple waves either (a) stick with numbering of questions even as you add or subtract or move questions between waves or (b) build a key file to let you aggregate across waves at the time the revised survey is being built. This will save months of work.

Most importantly, include each and every successive step in cleaning your data, using scripts located inside a "*build*" folder. When building a rectangular flatfile from any dataset many features that may have high missingness, create a full baseframe of id columns, and then join all values to that baseframe. Joining to this validated 'blank slate' dataframe removes the potential for errors or severely altering values. There can be several of these base dataframes at different levels.

Geospatial data may also require additional steps for accessible, reproducible research. In particular, when using GIS in R, converting ESRI shapefiles into SpatialPointsDataFrames and storing the result inside .rds files will significantly reduce long load times for future researchers. Filter large geospatial data sources to only necessary components whenever possible.

8. Analysis

Break the statistical analysis process into discrete stages, e.g.

1. Descriptive Statistics and Visualizations
2. Econometric Models
3. Figure Generation and Displaying Results

Script all figure and table generation, and avoid copying and pasting TeX text into editors. Instead generate and display well-formatted results using native R or Python. Whatever your approach to automation, keep it consistent throughout the project and document it inside a readme file inside the same folder as your analysis scripts.

Also script map generation whenever possible, particularly when you are tasked to generate several maps in sequence. Creating a set of reproducible scripted steps using R packages is preferable to relying on ArcGIS software, because R can implement the exact same spatial manipulations and also be ran anywhere sans licensing considerations. Though the initial learning curve is steeper, in R one can iterate very quickly, plotting a visual directly projecting coordinates on a scatter using ggplot, filtering these points by category, and then immediately execute spatial joins wherever relevant based on these same attributes.

9. Papers, Presentations, and Reports

It can be helpful to divide reports into *Internals* (within-colleagues) and *Externals* (to share or public-facing). Externals include codebooks that describe all data sources used, cleaning operations, field names, dates, and computational software packages.

Technical reports should be written in markdown of any kind, simply using the fewest, most human-readable tools. While scripts that generate figures and tables should go in the *Code* folder, the exported final deliverable and the individual plot images should be placed in (or exported to) the relevant *Reports* subfolder.

When compiling markdown documents, either use repeated 'source' calls to smaller individual files to create plot or model variables that will be included, or, store all of these variables, models, ggplots, and other R objects into a .rds file and instantly load them inside the markdown document for rapid presentation (or, for very large-data ggplots, just save the image and call the image file with a tag inside the .Rmd file). In particular, RMarkdown great for internal documents explaining code, process, and useful visuals. For papers being prepared for submission to journals, one can use R/Python pandoc packages, but raw LaTeX is likely more appropriate. In either case, one master file should call each individual section file, and sub-folders within the paper folder should store figures and internals. Here code modularity again serves dual purposes of navigability for collaborators, and ease of use for future researchers attempting to replicate your work.

For beamer/powerpoint/keynote –style presentations,

1. Always use letterbox format, no 4:3 aspect ratios.
2. Include a readme .txt file that tells future researchers which version of the paper figures and tables were pulled from if this is at all unclear. Keynote presentation can be generated in R or Python markdown formats, however, it is possible and should be analogous to working in PPT or Beamer.
3. For images that need to be cut and pasted, take high-resolution screenshots and insert into the software rather than pasting pdf files which do not render as well. If using PPT on a PC ensure you have set your default to high resolution, otherwise your edits will screw up your co-authors graphics.

Papers should be written in LaTeX (or LaTeX PDF generating R/Python source scripts, pandoc, etc.). One master file should call each individual section file, and sub-folders within the paper folder should store figures and internals. An example of such a master file is below.

```
\maketitle

\begin{abstract}
How natural disasters affect politics in developing countries is an important question given the fragility of fledgling democratic institutions in some of these countries as well as likely increased exposure to natural disasters over time due to climate change. Research in sociology and psychology suggests traumatic events can inspire pro-social behavior and therefore might increase political engagement. Research in political science argues that economic resources are critical for political engagement and thus the economic dislocation from disasters may dampen participation. We argue that when the government and civil society response effectively blunts a disaster's economic impacts, then political engagement may increase as citizens learn about government capacity. Using diverse data from the massive 2010-11 Pakistan floods, we find that Pakistanis in highly flood-affected areas turned out to vote at substantially  higher rates three years later than those less exposed. We also provide speculative evidence on the mechanism. The increase in turnout was higher in areas with lower {\it{ex ante}} flood risk, which is consistent with a learning process. These results suggest that natural disasters may not necessarily undermine civil society in emerging developing democracies. (JEL: A12, D72, D74, I28, O17)
\end{abstract}

\newpage
\doublespacing

\input{introduction_v21}

\input{background_v21}

\input{data_v21}

\input{empiricalstrategy_v21}

\input{results_v21}

\input{robustness_v21}

\input{conclusion_v21}

\clearpage
\singlespacing
\input{figures}

\clearpage
%\singlespacing
\input{tables}
%\input{tables_nostars}

\clearpage
%\singlespacing
\bibliography{jake,patrick,christine}

\clearpage
\appendix
\renewcommand\thefigure{\thesection.\arabic{figure}}
\renewcommand\thetable{\thesection.\arabic{table}}
\input{appendixA}

\clearpage
\setcounter{table}{0}
\renewcommand\thetable{\thesection.\arabic{table}}
\input{appendixB}
%\input{appendixB_nostars}

\clearpage
\setcounter{table}{0}
\renewcommand\thetable{\thesection.\arabic{table}}
\input{appendixC}
%\input{appendixC_nostars}

\end{document}
```

10. Meeting Notes

Keep consistent meeting notes throughout every project meeting with PIs and others during or immediately after, and clearly delegate who is responsible for this note keeping. Note should include date, subject, participants, and details of what was discussed.

11. Managing IP

Various projects often involve using data which are either confidential or are licensed based on a commercial agreement with limited usage rights. Do not host a data providers' IP on an insecure

cloud or online version control, this means **NOT GitHub**. Private GitHub accounts may be freely available for students / .edu email addresses, but as a general rule avoid GitHub anyway.

Use local version control (i.e. Git minus the 'Hub') and the author must never forget s/he cannot include these data with other public files. Establish a clear shared IP agreement early on between PIs and co-authors, and document if necessary.

12. Authorship and attribution

Authorship on large team projects is often a sticky issue. Three good informal rules are: (1) people should receive credit as an author if they make a major conceptual or data contribution to the project; and (2) senior folks should be generous in bringing junior folks on as co-authors and in author ordering decisions (e.g. if a PI is working on a project with a graduate student who is on the market, and the contributions have been close to equal, then make the student first author).

More formally, Miriam Golden lays out an excellent set of criteria in her 2018 note on co-authorship: "The standard criteria for authorship that is laid out in numerous professional guidelines going back nearly fifty years are:

1. participating in at least some aspect of the research process (research design, data collection, data analysis); and
2. writing or revising all or part of the paper and contributing significant intellectual content; and
3. critically reviewing the entire paper prior to submission and publication; and
4. publicly accepting responsibility for the paper and the results.

Note that all of these must obtain for an individual to receive designation as an author, and all persons who meet these criteria must receive recognition as authors. Persons involved in a research project who do not meet all four criteria should receive recognition in the acknowledgements."

Golden's guide goes into great detail on a range of issues around co-authorship and can be found here.

13. Task Management and Communication

For large long-term projects, do not rely on email alone for messaging, communication, and task assignment, and email attachments only for relevant interim project deliverables. To manage communication between more than 3 parties, use an online platform that assigns project-relevant tasks to specific people and aggregates all relevant comments underneath each labeled task. That said, use this system sparingly (i.e. for very major tasks, for more than a Google Doc of agenda items) to avoid wasted time checking a 2nd or 3rd digital platform. Use Jira or Asana.

Within a team project, share to the point of pain. Push information consistently, and ideally establish weekly calls to check in a project progress. Google Docs are well-suited to

creating meeting notes and objectives, written in parallel. Record a link to the document inside the project's Dropbox (or use Dropbox's native Google Doc - like document sharing).

Here are some other best practices:

- Keep a detailed README guide to directory structure –
    - Depending on project complexity this could sit at the top level or at least at the data analysis level.
    - This should contain a brief overview of what each of the sub directories contains. It may include pointers to the code files that have been used to generate these datasets.
    - The idea is to be able to link the code scripts to datasets. A few months down the line it's difficult to trace which exact script/version was used to generate output datasets. It is also useful to know at a high level which datsets contain what level of aggregation, specifications.
    - This is particularly helpful when integrating multiple datasets, or where different teams are creating and analyzing datasets.
- Weekly project calls are almost always a good idea.

Appendices

Appendix A: Example Software Bundles

RStudio – Context dependent. Before investing many man-hours writing code from scratch spend some time digging around available resources. Take packages with a grain of salt. Invest sometime reading documentation and what the package does exactly. Never rely on a package for complex analytics (e.g. up-sampling the minority class in an ML algorithm) without knowing what is going on under the hood. Some useful bundles for most types of analysis are

- *knitr* and *rmarkdown* for report generation
- ggplot, cowplot, gridBase, gridExtra, gridGraphics, Rcolorbrewer – for making good publication quality plots, integrating multiple ggplots, integrating plots and tables.
- sp, rdgal, ggmap- for geoprocessing.
- dplyr, data.table , sqldf – excellent resources for easy data manipulation, sqldf requires very basic sql knowledge. Data.table is a boon for extremely large data sets.
- tidy r – tidyverse is a collection of useful packages as they share common data representation and can use pipes.
- haven, readstata13, foreign, xlsx, readxl – useful to read + write Stata, SPSS, SAS, excel files in R. Many other options available.
- doParallel – for parallel processing in R. Can spread the code across multiple nodes/cores
- stringr –and base R functions cover almost all basic string processing.

- Lubridate – date/time processing
- Hmisc, reshape2 – and base R for making cross tabs, aggregates, summary tables and reshaping data.
- RCurl – communicating with HTTP servers, FTP downloads.
- Caret, h2o – too many ML packages to name here. But caret and h2o are great wrappers around many base packages which can implement almost all major algorithms.
- Shiny – turn any R code into an interactive web application
- Devtools – install packages directly from github, make your own packages
- Google integrated packages – Many google functionalities can carried over in R. Inlcuding google drive, sheets, all API functions, google trends, google visualizations etc.

Python – Context dependent again. Most researcher will come with their own preferences. For a beginner these libraries are useful:

- NumPy – Advanced math functionalities.
- SciPy – More math functionalities, transformations, optimizations, matrix algebra.
- Matplotlib – plotting publication quality figures.
- Nltk – String manipulation
- **IPython – Command shell for interactive programming with a notebook like structure. Super useful.**

Appendix B: Example Lab Diary

```
1/3 Beginning of project
       Write intersection of Gentzkow/Shapiro & USG docs

1/4 Mocked-up document comparing Gentzkow & Shapiro to USG doc

1/11 Jake 1st cut, Dustin revising

1/24 Feedback from Manu implemented,
         Dustin added Appendix on combining Git and Dropbox

2/24
       Adding the lesson to always start with a stable base frame when generating
       a rectangular dataset / flatfile. Otherwise screw up missing values.
              Changes reflected in edits to RPG_Outline.docx, VI. c. "Data
Preparation"

... Draft awaiting Jake's review ….

1/17 - Editing the prose
```

Appendix C: Example .tex Files

See above for general structure. Attached is an example with tables generated in code and table notes in the .tex file.

```
\section*{Tables}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% TABLES 1: Main result with different controls
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\begin{table}[!h]
	\centering
	\resizebox{\linewidth}{!}{
		\begin{threeparttable}
			\caption{Main Result with Different Controls for Past Turnout} \label{mainresult}
			\begin{tabular}{l*{9}{c}}
				\input{TexTables/t1}
			\end{tabular}
			\begin{tablenotes}
				\item \scriptsize \emph{Notes}: Outcome variable is turnout in the 2013 election. Models 4 through 6 control for previous turnout using a trend variable ($trend=
\text{turnout}_{08} - \text{turnout}_{02}$).  Models 7 through 9 control for previous turnout through 2002 and 2008 turnout level variables ($\text{turnout}_{02}$, $\text{turnout}_{08}$). All
regressions include controls for ex ante UNEP flood risk, distance to major river, dummy for constituencies bordering a major river, std. dev. of the constituency's elevation, and mean
constituency elevation, as well as the percentage of the population affected by flooding in 2012. Unit of observation is a Provincial Assembly constituency.  Standard errors are clustered at the
district level and reported in parentheses. * p$<$0.10, ** p$<$0.05, *** p$<$0.01
			\end{tablenotes}
		\end{threeparttable}}
	\end{table}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% TABLES 2: Main result by different subsets
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\quad\\
\quad\\
\quad\\
\quad\\
\quad\\
\begin{table}[!h]
	\centering
	\resizebox{\linewidth}{!}{
		\begin{threeparttable}
			\caption{Main Result for Different Subsets}  \label{mainresult_subsets}
			\begin{tabular}{l*{9}{c}}
				\input{TexTables/t2}
			\end{tabular}
			\begin{tablenotes}
				\item \scriptsize \emph{Notes}: Outcome variable is turnout in the 2013 election. Unit of observation is a Provincial Assembly constituency.  All models control for previous
turnout through 2002 and 2008 turnout level variables ($\text{turnout}_{02}$, $\text{turnout}_{08}$). All regressions include controls for ex ante UNEP flood risk, distance to major river, dummy
for constituencies bordering a major river, std. dev. of the constituency's elevation, mean constituency elevation, and the percentage of the population affected by flooding in 2012. Standard
errors are clustered at the district level and reported in parentheses. * p$<$0.10, ** p$<$0.05, *** p$<$0.01
			\end{tablenotes}
		\end{threeparttable}}
	\end{table}
```

Appendix D: Integrating Dropbox and Version Control

The following example demonstrates how to set up git with Dropbox. As per a [popular StackOverflow question](), one way to use version control without exposing private data to public repositories online is to use Git with Dropbox.

> Within R Studio, which has it's own Terminal tab, first navigate to your project directory (in this example named ~/project , where "~" is shorthand for the home folder on Unix systems (this would be "C://Users/currentuser" in Windows).
>
> Then type the following commands (where everything before the '$' just labels your current directory location).

~/project $ git init

- Initializes git

~/project $ git add .

- Aggregates relevant files

~/project $ git commit -m "first commit"

- "Commits" or records your intended actions

~/project $ cd ~/Dropbox/git

- Changes your directory location to a Dropbox folder you created and named "git"

~/Dropbox/git $ git init --bare project.git

- Initializes empty project inside this folder

~/Dropbox/git $ cd ~/project

- Navigates back to your project's directory location

~/project $ git remote add origin ~/Dropbox/git/project.git

- Links your project folder contents to your Dropbox git folder

~/project $ git push -u origin master

- "Pushes" or finalizes your intended "commit" (revisions), saving your version control system to the Dropbox folder

Within your project, a particularly user-friendly way to use git is through RStudio. Inside the newer versions of RStudio, you can create an RStudio project in your existing project directory. Once this RStudio project is loaded, you can edit preferences to verify your project is linked to git, and make edits to your code using a tab named "Git". Here you can create commits and push your intended permanent revisions. Remember that all commits must include a comment, and are not permanent (or rather, made into the current code revision) until pushed.

Appendix E - Using Princeton's High-Speed Computing Clusters

Researchers have multiple university resources to make analysis easier and faster especially when dealing with extremely large datasets, or code that might take a long while to run on local machines. Typically, you will need PI approval to be added to the clusters.

Note – You may not need approval to be added to a smaller cluster (e.g. "adroit" at Princeton) which is ideal for prototyping code, learning to use the clusters etc.

Once added the following steps will help you get started – if the steps below are unclear there are detailed explanations for each of these on - https://researchcomputing.princeton.edu/help-and-faqs

1. **Install putty** – you will connect to the della/adroit clusters using SSH.
2. **Install Filezilla** to put your scripts/ code on the cluster or download the data+results from the cluster.
3. In this directory you will find a sample SLURM script to get you started. This is a super basic script that just helps define the number of nodes, number of cores, and the run time (approx) of your code. This script includes lines that send out an email once the job starts and ends. If the job fails you'll know too. Output dumps are saved in the SLURM directory itself. Good for job monitoring.
4. You can find the structure of della here - https://researchcomputing.princeton.edu/systems-and-services/available-systems/della. This will help you deicde what kind of resources you want for your job.
   It's obviously not wise to ask for more than 20 cores/node (longer wait time if you ask for 32 cores as only some nodes have that particular processor)
5. **Parallel computing** – There are a number of ways to parallelize a job. You can multithread in python/R or you can use slurm to split your job and run it on many different nodes/cores. Run some benchmarking tests to see what works best for you. Here are more SLURM resources, plus the internet is full of them. (https://researchcomputing.princeton.edu/education/online-tutorials)
6. Finally once you have your python code and slurm script ready submit your job with #SBATCH test.slurm.

Appendix F: Helpful R and Python Reference

General R:

| R for Data Science

| StackOverflow R Tag Info Page

| StackOverflow R Tag

| StackOverflow R-faq Tag

| Advanced R

| RStudio Cheatsheets

Search tags in StackOverflow w '[tag]'. Search question + tag before Google.

Ideal: Create StackOverflow account (even if you ask nothing) so you can star/favorite questions. Most are not so easily re-googled.

Spatial R:

| CRAN Spatial View

| CRAN Spatial Temporal View

| GIS StackExchange

| Nick Eubank GIS in R Tutorials

Recent Stanford “Data Challenge Lab” course curriculum. Great source to learn all R techniques using the best available resources and newest packages, from munging to spatial display. Co-taught by creator of dplyr and ggplot2. https://dcl-2017-01.github.io/curriculum/