February 2016.

The Plain Person's Guide to Plain Text Social Science[*]

Kieran Healy
Duke University
kjhealy@soc.duke.edu

Abstract: As a beginning graduate student in the social sciences, what sort of software should you use to do your work? More importantly, what principles should guide your choices? This article offers some answers. The short version is: you should use tools that give you more control over the process of data analysis and writing. I recommend you write prose and code using a good text editor; analyze quantitative data with R or Stata; minimize error by storing your work in a simple format (plain text is best), and make a habit of documenting what you've done. For data analysis, consider using a format like Rmarkdown and tools like Knitr to make your work more easily reproducible for your future self. Use Pandoc to turn your plain-text documents into PDF, HTML, or Word files to share with others. Keep your projects in a version control system. Back everything up regularly. Make your computer work for you by automating as many of these steps as you can. To help you get started, I briefly discuss a drop-in set of useful defaults to get started with Emacs (a powerful, free text-editor). I share some templates and style files that can get you quickly from plain text to various output formats. And I point to several alternatives, because no humane person should recommend Emacs without presenting some other options as well.

INTRODUCTION

You can do productive, maintainable and reproducible work with all kinds of different software set-ups. This is the main reason I don't go around encouraging everyone to convert to the applications I use. (My rule is that I don't try to persuade anyone to switch if I can't commit to offering them technical support during and after their

[*]This is an updated and expanded version of 'Choosing Your Workflow Applications' (2013). The main difference is an increased emphasis on Rmarkdown, `knitr`, and `pandoc`. The `.md` source for this file is available at `https://github.com/kjhealy/workflow-paper`. It is also available as a website, at `http://plain-text.co`.

move.) So this discussion is not geared toward convincing you there is One True Way to organize things. I do think, however, that if you're in the early phase of your career as a graduate student in, say, Sociology, or Economics, or Political Science, you should give some thought to how you're going to organize and manage your work.[1] This is so for two reasons. First, the transition to graduate school is a good time to make changes. Early on, there's less inertia and cost associated with switching things around than there will be later. Second, in the social sciences, text and data management skills are usually not taught to students explicitly. This means that you may end up adopting the practices of your advisor or mentor, continue to use what you are taught in your methods classes, or just copy whatever your peers are doing. Following these paths may lead you to an arrangement that you will be happy with. But maybe not. It's worth looking at the options.

Two remarks at the outset. First, because this discussion is aimed at beginning students, some readers may find much with which they are already familiar. Even so, some sections may still be of interest, as I have tried to keep the software references current. Second, although in what follows I advocate you take a look at several applications in particular, it's not really about the gadgets or utilities. The Zen of Organization is not to be found in Fancy Software. Nor shall the true path of Getting Things Done be revealed to you through the purchase of a nice Moleskine Notebook. Instead, it lies within—unfortunately.

TWO REVOLUTIONS IN COMPUTING

When talking to undergraduates or graduate students on this topic, and when teaching classes that use these tools, I increasingly run into the problem that it's hard to get started without backing up a bit first in order to talk about how the computer they are using works. I think the reason for this is the rise of the flat-screen, touch-based model of computing, most obviously on phones and then very secondarily on things like Apple's iPad or Microsoft's Surface tablet. Now, most people who need to write long documents (like papers or dissertations) or work in an involved way with data do not use a tablet as their primary device. But it does seem clear that some kind of touch-screen interaction is the future of computing for most people. Indeed, once you consider phones properly you realize it's the *present* of computing for most people.

While it is not strictly impossible, it remains very difficult to do your academic, social-science work on a device of this sort. This is likely to be the case for some time, as the tools we have are not designed up for them. That's not surprising, but I think there is an underappreciated tension here. Two ongoing computing revolutions are tending

[1]This may also be true if you are about to move from being a graduate student to starting as a faculty member, though perhaps the rationale is less compelling given the costs.

to pull in opposite directions. On one side, the mobile, cloud-centered, touch-screen, phone-or-tablet model has brought powerful computing to more people than ever before. This revolution is the one everyone is talking about, because it is happening on a huge scale and is where all the money is. In practice it puts single-purpose applications in the foreground and hides from the user both the workings of the operating system and (especially) the structure of the file system where items are stored and moved around.

On the other side, open-source tools for plain-text coding, data analysis, and writing are also better and more accessible than they have ever been. This has happened on a smaller scale than the first revolution, of course. But still, these tools really have revolutionized the availability and practice of data analysis and scientific computing generally. They continue to do so, too, as people work to make them better at everything from slurping up data on the web to presenting it there. These tools mostly work by gluing together separate, specialized widgets into a reproducible workflow. They are "bitty" or granular because the process of data analysis is that way as well. They do much less to hide the operating system layer—instead they often directly mesh with it—and they also presuppose a working knowledge of the file system underpinning the organization of the things the researcher is using or creating, from data files to code to figures and final papers.

The tension is that, increasingly, people—people like the target audience of this article—entering the world of social science wanting to work with data tend to have little or no prior experience with text-based, command-line, file-system-dependent tools. In many cases, they do not have much experience making effective use of a multi-tasking windowing environment, either, at least in the sense of making applications work together in the service of a single goal.[2] To be clear, this is not something to blame users for. Neither is it some misguided nostalgia on my part for the command line. Rather, it is an aspect of how computer use is changing at a very large scale. The coding and data analysis tools we have are powerful and for the most part meant to allow research products to be opened up and inspected. But the way they work clearly runs against the current of everyday, end-use computing, which increasingly hides many implementation details and focuses on single-purpose tasks. Again, specialized tools are necessarily specialized. The net result for the social sciences in the short to medium term, I think, is that we will have a suite of powerful tools that enable an amazing variety of scientific activity, developed in the open and mostly available for free. But it will get harder to teach people how to use them, and perhaps even to convince people to try them.

[2] As opposed to multi-tasking in the less-interesting sense of trying to pay attention to a number of discrete tasks (writing, email, calendar, web-browsing), each controlled by a separate application.

WHAT'S THE PROBLEM?

The problem is that doing scholarly work is intrinsically a mess. There's the annoying business of getting ideas and writing them down, of course, but also everything before, during, and around it: data analysis and all that comes with it, and the tedious but unavoidable machinery of scholarly papers—especially citations and references. There is a lot of keep track of, a lot to get right, and a lot to draw together at the time of writing. Academic papers are by no means the only form of writing subject to constraints of this sort. Consider this sensible discussion by Dr. Drang, a (pseudonymous) consulting engineer:

> I don't write fiction, but I can imagine that a lot of fiction writing can be done without any reference materials whatsoever. Similarly, a lot of editorials and opinion pieces are remarkably fact-free; these also can spring directly from the writer's head. But the type of writing I typically do—mostly for work, but also here—is loaded with facts. I am constantly referring to photographs, drawings, experimental test results, calculations, reports written by others, textbooks, journal articles, and so on. These are not distractions; they are essential to the writing process.
>
> And it's not just reference material. Quite often I need to make my own graphs and drawings to include in a report. Because the text and the graphics are all part of a coherent whole, I need to go back and forth between the two; the words inform the pictures and the pictures inform the words. This is not the Platonic ideal of a clean writing environment—a cup of coffee on an empty desk in a white room—that you see in videos for distraction-free editors.
>
> Some of the popularity of these editors is part of the backlash against multitasking, but people are confusing themselves with their computers. When I'm writing a report, that is my single task, and I bring to bear whatever tools are necessary to complete it. That my computer is multitasking by running many programs simultaneously isn't a source of confusion or distraction, it's the natural and efficient way for me to get my one task done.

A lot of academic writing is just like this. It can be tricky to manage. It's even worse when you have collaborators and other contributors. So, what to do?

THE OFFICE MODEL AND THE ENGINEERING MODEL

Let me make a crude distinction. There are "Office Type" solutions to this problem, and there are "Engineering Type" solutions. Don't get hung up on the distinction or the labels. Office solutions tend towards a cluster of tools where something like Microsoft Word is at the center of your work. A Word file or set of files is the most "real" thing in your project. Changes to your work are tracked inside that file or files. Citation and reference managers plug into them. The outputs of data analyses—tables, figures—get dropped into them or kept alongside them. The master document may be passed around from person to person or edited and updated in turn. The final output is exported from it, perhaps to PDF or to HTML, but maybe most often the final output just *is* the `.docx` file, cleaned up and with the track changes feature turned off.

In the Engineering model, meanwhile, plain text files are at the center of your work. The most "real" thing in your project will either be those files or, more likely, the Git repository that controls the project. Changes are tracked outside the files. Data analysis is managed in code that produces outputs in (ideally) a known and reproducible manner. Citation and reference management will likely also be done in plain text, as with a BibTeX `.bib` file. Final outputs are assembled from the plain text and turned to `.tex`, `.html`, or `.pdf` using some kind of typesetting or conversion tool. Very often, because of some unavoidable facts about the world, the final output of this kind of solution is also a `.docx` file.

This distinction is meant to capture a tendency in organization, not a rigid divide—still less a sort of personality. Obviously it is possible organize things on the Office model. (Indeed, it's the dominant way.) Applications like Scrivener, meanwhile, combine elements of the two approaches. Scrivener embraces the "bittyness" of large writing projects in an effective way, and can spit out clean copy in a variety of formats. Scrivener is built for people writing lengthy fiction (or qualitative non-fiction) rather than anything with quantitative data analysis, so I have never used it extensively. Microsoft Word, meanwhile, still rules large swathes of the Humanities and the Social Sciences, and the production process of many publishers. So even if you prefer plain text for other reasons—especially in connection with project management and data analysis—the routine need or obligation to provide a Word document to someone is one of the main reasons to want to be able to easily convert things. HTML is a great lingua franca.

This article is mostly about the Engineering model. But many people use the Office model, and you may end up working with (or for) some of them. In those cases, it is generally easier for you to use their software than *vice versa*, if only because you are likely have a copy of Word on your computer already. In these circumstances you might also collaborate using Google Docs or some other service that allows for simultaneously editing the master copy of a document. This may not be ideal, but it is better than not collaborating. There is little to be gained from plain-text dogmatism in a `.docx` world.

MAKE SURE YOU KNOW WHAT YOU DID

For any kind of formal data analysis that leads to a scholarly paper, whichever model you tend to favor, there are some basic principles to adhere to. Perhaps the most important thing is to do your work in a way that leaves a coherent record of your actions. Instead of doing a bit of statistical work and then just keeping the resulting table of results or graphic that you produced, for instance, write down what you did as a documented piece of code. Rather than figuring out but not recording a solution to a problem you might have again, write down the answer as an explicit procedure. Instead of copying out some archival material without much context, file the source properly, or at least a precise reference to it.

A second principle is that a document, file or folder should always be able to tell you what it is. Beyond making your work reproducible, you will also need some method for organizing and documenting your draft papers, code, field notes, datasets, output files or whatever it is you're working with. In a world of easily searchable files, this may mean little more than keeping your work in plain text and giving it a descriptive name. It should generally *not* mean investing time creating some elaborate classification scheme or catalog that becomes an end in itself to maintain.

A third principle is that repetitive and error-prone processes should be automated if possible. (Software developers call this "DRY", or Don't Repeat Yourself.) This makes it easier to check for and correct mistakes. Rather than copying and pasting code over and over to do basically the same thing to different parts of your data, write a general function that can be called whenever it's needed. Instead of retyping and reformatting the bibliography for each of your papers as you send it out to a journal, use software that can manage this for you automatically.

There are many ways of implementing these principles. You could use Microsoft Word, Endnote and SPSS. Or Textpad and Stata. Or a pile of legal pads, a calculator, a pair of scissors and a box of file folders. But software applications are not all created equal, and some make it easier than others to do the Right Thing. For instance, it is *possible* to produce well-structured, easily-maintainable documents using Microsoft Word. But you have to use its styling and outlining features strictly and responsibly, and most people don't bother. You can maintain reproducible analyses in SPSS, but the application isn't set up to do this automatically or efficiently, nor does its design encourage good habits. So, it is probably a good idea to invest some time learning about the alternatives. Many of them are free to use or try out, and you are at a point in your career where you can afford to play with different setups without too much trouble.

The dissertation, book, or articles you write will generally consist of the main text, the results of data analysis (perhaps presented in tables or figures) and the scholarly apparatus of notes and references. Thus, as you put a paper or an entire dissertation together you will want to be able to easily *keep a record of your actions* as you *edit text*, *analyze data* and *present results* in a *reproducible way*. In the next section I describe some applications and tools designed to let you do all of this. I focus on tools that fit together well (by design) and that are all freely available for Windows, Linux and Mac OS X. They are not perfect, by any means—in fact, some of them can be awkward to learn. But graduate-level research and writing can also be awkward to learn. Specialized tasks need specialized tools and, unfortunately, although they are very good at what they do, these tools don't always go out of their way to be friendly.

Use version control

Writing involves a lot of editing and revision. Data analysis involves cleaning files, visualizing information, running models, and repeatedly re-checking your code for mistakes. You need to keep track of this work. As projects grow and change, and as you explore different ideas or lines of inquiry, the task of documenting your work at the level of particular pieces of code or edits to paragraphs in individual files can become more involved over time. The best thing to do is to institute some kind of *version control* to keep a complete record of changes to a single file, a folder of material, or a whole project. A good version control system allows you to "rewind the tape" to earlier incarnations of your notes, drafts, papers and code. It lets you keep explore different aspects or branches of a project. In its more developed forms it provides you with some powerful tools for collaborating with other people. And it helps stop you from having directories full of files with confusingly similar names like `Paper-1.doc`, `Paper-2.doc`, `Paper-conferenceversion.doc`, `Paper-Final-revised-DONE-lastedits.doc`.

In the social sciences and humanities, you are most likely to have come across the idea of systematic version control by way of the "Track Changes" feature in Microsoft Word, which lets you see the edits you and your collaborators have made to a document. Collaborative editing of a single document is also possible through platforms like Google Docs or Quip. True version control is a way to do these things for whole projects, not just individual documents, in a comprehensive and transparent fashion. Modern version control systems such as Mercurial and Git can, if needed, manage very large projects with many branches spread across multiple users. Git has become the *de facto* standard, and GitHub is a place where software developers and social scientists make their work available, and where you can contribute to ongoing projects or make public your own.

Modern version control requires getting used to some new concepts related to tracking your files, and learning how your version control system implements these concepts. There are some good resources for learning them. Because of their power, these tools might seem like overkill for individual users. (Again, though, many people find Word's "Track Changes" feature indispensable once they begin using it.) But version control systems can be used quite straightforwardly in a basic fashion, and they can often be easily integrated with your text editor, or used via a friendlier application interface that keeps you away from the command line. The core idea is shown in Figure 1. You keep your work in a *repository*. This can be kept locally, or on a remote server. As you work, you periodically *stage* your changes and then *commit* them to the repository, along with a little note about what you did. Repositories can be copied, cloned, merged, and contributed to by you or other people.

Revision control has significant benefits. A tool like Git combines the virtues of "track changes" with those of backups. Every repository is a complete, self-contained, cryptographically signed copy of the project, with a log of every recorded step in its development by all of its participants. It puts you in the habit of committing changes to a file or project piecemeal as you work on it, and (briefly) documenting those changes as you go. It allows you to easily test out alternative lines of development or thinking by creating "branches" of a project. It allows collaborators to work on a project at the same time without sending endless versions of the "master" copy back and forth via email. And it provides powerful tools that allow you to automatically merge or (when necessary) manually compare changes that you or others have made. Perhaps most importantly, it lets you revisit any stage of a project's development at will and reconstruct what it was you were doing. This can be useful whether you are writing code for a quantitative analysis, managing field notes, or writing a paper. While you will probably not need to control everything in this way, I *strongly* suggest you consider managing at least the core set of text files that make up your project (e.g., the code that does the analysis and generates your tables and figures; the dataset itself; your notes and working papers, the chapters of your dissertation, etc). As time goes by you will generate an extensive, annotated record of your actions that is also a backup of your project at every stage of its development. Services such as GitHub allow you to store public or (for a fee) private project repositories and so can be a way to back up work offsite as well as a platform for collaboration and documentation of your work.

Why should you bother to do any of this? Because the main person you are doing it for is *you*. Papers take a long time to write. When you inevitably return to your table or figure or quotation nine months down the line, your future self will have been saved hours spent wondering what it was you thought you were doing and where you got it from.

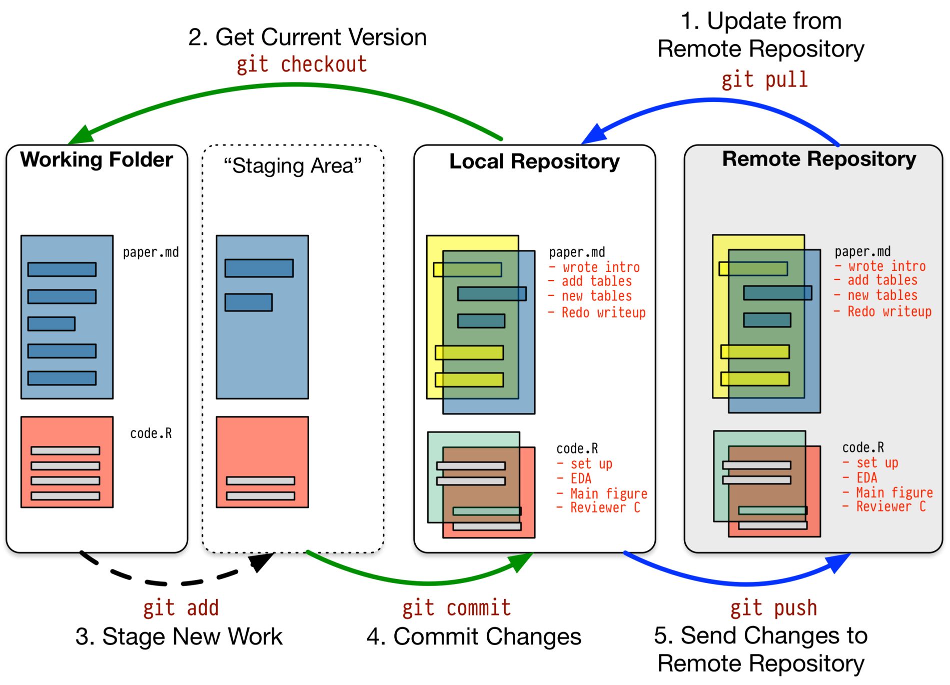


Figure 1: A schematic `git` workflow. (1) You get the most recent version of your project from a remote repository (such as GitHub), and then (2) You "check-out" the project to work on, writing text or code, etc. You work on files as usual in a folder on your computer. But it is the repository that is "real" as far as the project is concerned. (3) You edit your documents, and once you are happy with the changes you *stage* them. Behind the scenes means the changes are added to an index that Git uses to keep track of things. But these changes are not yet permanently recorded. (4) You then *commit* the changes to the repository, along with a note about what you did. This is now a firm record of the changes. (5) You "push" the changes up to the remote repository, which in effect also functions as a backup of your work. Over time, the repository comes to contain a complete record of the project, any step of which can be revisited as needed. In the simplest case there is no remote repository, only a local one you check out and commit changes to. You can do all this from the command line, or use one of several front-end applications designed to help you manage things.

Back up your work

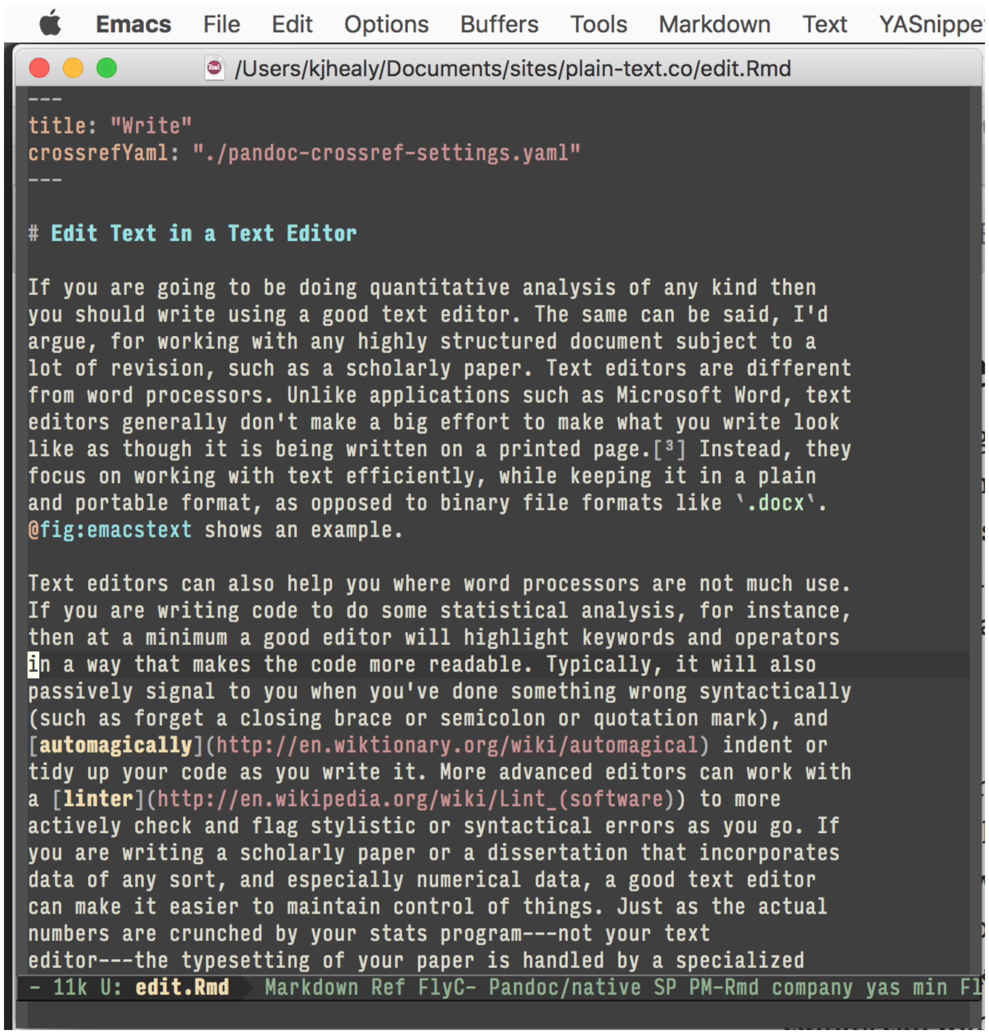
Regardless of whether you choose to use a formal revision control system, you should nevertheless have *some* kind of systematic method for keeping track of versions of your files. Version-controlled projects are backed-up to some degree if you keep your repository somewhere other than your work computer. But this is not nearly enough. Apple's Time Machine software, for example, backs up and versions your files on your disk, or to a local hard drive, allowing you to step back to particular instances of the file you want. This still isn't enough, though. You need regular, redundant, automatic, off-site backups of your work. Because you are lazy and prone to magical thinking, you will not do this responsibly by yourself. This is why the most useful backup systems are the ones that require a minimum amount of work to set up and, once organized, back up everything automatically without you having to remember to do anything. This means paying for a secure, offsite backup service like Crashplan, or Backblaze. Offsite backup means that in the event (unlikely, but not unheard of) that your computer *and* your local backups are stolen or destroyed, you will still have copies of your files. I know of someone whose office building was hit by a tornado. She returned to find her files and computer sitting in a foot of water. You never know. Less dramatically, but no less catastrophic from the point of view of one's work, I know people who have lost months or even years of work as a result of dropping a laptop, or having it stolen, or simply having their computer (or "backup drive") fail for no apparent reason. Like seat belts, you don't need backups until you really, really need them. As Jamie Zawinski has remarked, when it comes to losing your data "The universe tends toward maximum irony. Don't push it."

EDIT TEXT

Use a Text Editor

If you are going to be doing quantitative analysis of any kind then you should write using a good text editor. The same can be said, I'd argue, for working with any highly structured document subject to a lot of revision, such as a scholarly paper. Text editors are different from word processors. Unlike applications such as Microsoft Word, text editors generally don't make a big effort to make what you write look like as though it is being written on a printed page.[3] Instead, they focus on working with text efficiently, while keeping it in a plain and portable format, as opposed to binary file formats like `.docx`. Figure 2 shows an example.

[3]For further argument about the advantages of text-editors over word processors see Allin Cottrell's polemic, "Word Processors: Stupid and Inefficient."


Emacs File Edit Options Buffers Tools Markdown Text YASnippe
/Users/kjhealy/Documents/sites/plain-text.co/edit.Rmd

```
---
title: "Write"
crossrefYaml: "./pandoc-crossref-settings.yaml"
---

# Edit Text in a Text Editor

If you are going to be doing quantitative analysis of any kind then
you should write using a good text editor. The same can be said, I'd
argue, for working with any highly structured document subject to a
lot of revision, such as a scholarly paper. Text editors are different
from word processors. Unlike applications such as Microsoft Word, text
editors generally don't make a big effort to make what you write look
like as though it is being written on a printed page.[^3] Instead, they
focus on working with text efficiently, while keeping it in a plain
and portable format, as opposed to binary file formats like `.docx`.
@fig:emacstext shows an example.

Text editors can also help you where word processors are not much use.
If you are writing code to do some statistical analysis, for instance,
then at a minimum a good editor will highlight keywords and operators
in a way that makes the code more readable. Typically, it will also
passively signal to you when you've done something wrong syntactically
(such as forget a closing brace or semicolon or quotation mark), and
[automagically](http://en.wiktionary.org/wiki/automagical) indent or
tidy up your code as you write it. More advanced editors can work with
a [linter](http://en.wikipedia.org/wiki/Lint_(software)) to more
actively check and flag stylistic or syntactical errors as you go. If
you are writing a scholarly paper or a dissertation that incorporates
data of any sort, and especially numerical data, a good text editor
can make it easier to maintain control of things. Just as the actual
numbers are crunched by your stats program---not your text
editor---the typesetting of your paper is handled by a specialized
- 11k U: edit.Rmd  Markdown Ref FlyC- Pandoc/native SP PM-Rmd company yas min Fl
```

Figure 2: Working on part of this document in Emacs.

Text editors can also help you where word processors are not much use. If you are writing code to do some statistical analysis, for instance, then at a minimum a good editor will highlight keywords and operators in a way that makes the code more readable. Typically, it will also passively signal to you when you've done something wrong syntactically (such as forget a closing brace or semicolon or quotation mark), and automagically indent or tidy up your code as you write it. More advanced editors can work with a linter to more actively check and flag stylistic or syntactical errors as you go. If you are writing a scholarly paper or a dissertation that incorporates data of any sort, and especially numerical data, a good text editor can make it easier to maintain control of things. Just as the actual numbers are crunched by your stats program—not your text editor—the typesetting of your paper is handled by a specialized application, too. That tool should automatically take care of things like entries in your bibliography, the labelling of tables and figures, and cross-references and other paraphernalia. The best editors can closely integrate with the tools you use to do the various pieces of your work.

Emacs is a text editor, in the same way the blue whale is a mammal. It does the things I have just described, and rather more besides, if you want it to. Combining Emacs with some other applications and add-ons allows you to manage writing and data-analysis effectively. If it seems odd to do a bunch of different tasks inside an editor, the blogger Rekado makes a useful analogy to the way people use web browsers:

> Just like a browser is used by many as a platform for running applications operating on some HTML document, Emacs is a platform for anything that can "reasonably" (this is up for interpretation) be mapped to buffers of text. Applications in browsers are written in JavaScript, applications in Emacs are written in EmacsLisp (also called "elisp"). … If you have used your web browser (or have observed someone use their web browser) to play games, listen to music, watch videos, read and compose email, edit text (e.g. by contributing to the Wikipedia), chat with friends (or chat about foes), read documentation, installed an extension—well, then the notion of a generic tool as a platform should not be a foreign concept to you. Emacs can be understood as such a generic tool providing a text interface (one of which may be a file editor).

While very powerful and flexible, Emacs can be annoying. Indeed, to many people encountering it for the first time—especially those used to standard applications on Windows or Mac OS—its conventions seem bizarre and byzantine. As applications go, Emacs is quite ancient. The first version was written by Richard Stallman in the 1970s. Because it evolved in a much earlier era of computing (before the development of decent graphical displays, for instance, and window managers, and possibly also fire), it doesn't share many of the conventions of modern applications. Like most powerful text editors, Emacs offers many opportunities to waste your time learning its particular conventions, tweaking its settings, and generally customizing it. There are several good alternatives on each major platform, and I discuss some of them below.

Given all that, why mention it in the first place? Partly because it's the editor I use. Partly because it is available for all of the main desktop and laptop computing plaforms. And partly becuase it is very, *very* good at doing what I want it to do. There are many good reasons to use something like TextMate, or Sublime Text instead of Emacs (I return to these alternatives below). Similarly, when doing data analysis with R, you may just want to use the RStudio environment. You will do fine if you prefer these alternatives. You will do fine if you go with these alternatives.

Write in Markdown

When you write papers in plain text, how do you manage the formatting, sectioning, and other related aspects of your document? Markdown is a loosely-standardized way of writing plain text that includes information about the formatting of your document. It was originally developed by John Gruber, with input from Aaron Swartz. The aim was to make a simple format that could incorporate structural information about the document (such as headings and subheadings, *emphasis*, hyperlinks, lists, footnotes, and so on), with minimal loss of readability. Formats like HTML or TeX are much more extensive markup languages, but Markdown was meant to be simple. Over the years it has become a *de facto* standard. Text editors and note-taking applications support it, and tools exist to convert Markdown not just into HTML (its original target output format) but many other document types as well. Listing 1 shows the markdown source for this paragraph and its subheading.

Listing 1: The Markdown source for a nearby part of this document.

```
When you write papers in plain text, how do you manage the formatting,
sectioning, and other related aspects of your document?
[Markdown](http://en.wikipedia.org/wiki/Markdown) is a loosely-standardized
way of writing plain text that includes information about the formatting of
your document. It was originally developed by John Gruber, with input
from Aaron Swartz. The aim was to make a simple format that could
incorporate structural information about the document (such as
headings and subheadings, *emphasis*,
[hyperlinks](http://daringfireball.net/markdown), lists, footnotes,
and so on), with minimal loss of readability. Formats like HTML or TeX
are much more extensive markup languages, but Markdown was meant to be
simple. Over the years it has become a *de facto* standard. Text
editors and note-taking applications support it, and tools exist to
convert Markdown not just into HTML (its original target output
format) but many other document types as well. @lst:markdown-example shows
the markdown source for this paragraph and its subheading.
```

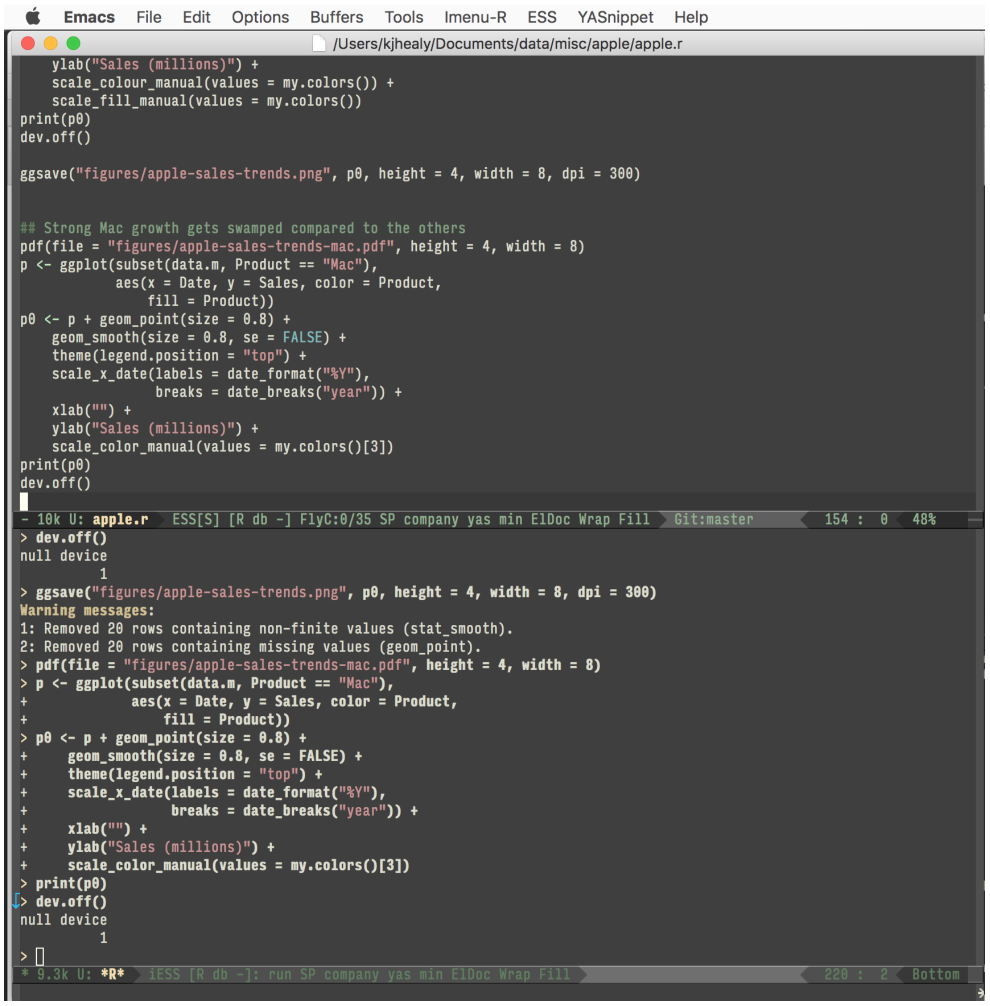
The excerpt shown in Listing 1 shows a few of the most common Markdown conventions, most notably how it represents headings and subheadings (a `#` symbol for a top-level header, with `##` for the next level down, and so on), how it represents hyperlinks, and how it emphasizes text. There are a number of Markdown variants, or "flavors", that have extended it to manage things like cross-references and labels, citations, and other textual elements. Citations are particularly important. The `pandoc-citeproc` filter is an add-on that handles these. It can be installed alongside `pandoc`. Your bibliography can be stored in one of a variety of formats (such as BibTeX, or EndNote). Within your `.md` document, cites are referred to by their key, such as `[@healy14datavisualsociol]`. When `pandoc` converts your document, the cite key is replaced with the reference information like this (Healy and Moody 2014), and the full bibliographic entry is included in an automaticaly-generated list of references. Read Pandoc's documentation for more details about citations. At the end of the excerpt you can also see that the code listing is labeled with `@lst:markdown-example`, for example. A Pandoc filter named `pandoc-crossref` extends this `@label` convention to deal with labeled Figures, Tables, and so on. Using Markdown in this way means you do not have to worry whether your reference list is complete, or whether cross-references (to 'Figure 3' for example) remain correct after you move things around in your text.

Use R with ESS or RStudio

You will probably be doing some—perhaps a great deal—of quantitative data analysis. R is an environment for statistical computing. It's well-supported, continually improving, and has a very active expert-user community. The documentation that comes with the software is complete, if somewhat terse, but there are a large number of excellent reference and teaching texts that illustrate its use. These include Dalgaard (2008), Venables and Ripley (2002), Maindonald and Braun (2003), Fox (2002), Harrell (2016), Matloff (2011), and Gelman and Hill (2007). Although it is a command-line tool at its core, it can easily be used in conjunction with the RStudio IDE. You can download R from The R Project Homepage.

R can be used directly within Emacs by way of a package called ESS (for "Emacs Speaks Statistics"). As shown in Figure 3, it allows you to work with your code in one Emacs frame and a live R session in another right beside it. Because everything is inside Emacs, it is easy to do things like send a chunk of your code over to R using a keystroke. This is a very efficient way of doing interactive data analysis while building up code you can use again in future.

You'll present your results in papers, but also in talks where you will likely use some



```
    ylab("Sales (millions)") +
    scale_colour_manual(values = my.colors()) +
    scale_fill_manual(values = my.colors())
print(p0)
dev.off()

ggsave("figures/apple-sales-trends.png", p0, height = 4, width = 8, dpi = 300)


## Strong Mac growth gets swamped compared to the others
pdf(file = "figures/apple-sales-trends-mac.pdf", height = 4, width = 8)
p <- ggplot(subset(data.m, Product == "Mac"),
            aes(x = Date, y = Sales, color = Product,
                fill = Product))
p0 <- p + geom_point(size = 0.8) +
    geom_smooth(size = 0.8, se = FALSE) +
    theme(legend.position = "top") +
    scale_x_date(labels = date_format("%Y"),
                 breaks = date_breaks("year")) +
    xlab("") +
    ylab("Sales (millions)") +
    scale_color_manual(values = my.colors()[3])
print(p0)
dev.off()

- 10k U: apple.r  ESS[S] [R db -] FlyC:0/35 SP company yas min ElDoc Wrap Fill  Git:master      154 :  0  48%
> dev.off()
null device
          1
> ggsave("figures/apple-sales-trends.png", p0, height = 4, width = 8, dpi = 300)
Warning messages:
1: Removed 20 rows containing non-finite values (stat_smooth).
2: Removed 20 rows containing missing values (geom_point).
> pdf(file = "figures/apple-sales-trends-mac.pdf", height = 4, width = 8)
> p <- ggplot(subset(data.m, Product == "Mac"),
+             aes(x = Date, y = Sales, color = Product,
+                 fill = Product))
> p0 <- p + geom_point(size = 0.8) +
+     geom_smooth(size = 0.8, se = FALSE) +
+     theme(legend.position = "top") +
+     scale_x_date(labels = date_format("%Y"),
+                  breaks = date_breaks("year")) +
+     xlab("") +
+     ylab("Sales (millions)") +
+     scale_color_manual(values = my.colors()[3])
> print(p0)
> dev.off()
null device
          1
> 
* 9.3k U: *R*  iESS [R db -]: run SP company yas min ElDoc Wrap Fill                        220 :  2  Bottom
```

Figure 3: Working with R in Emacs using ESS. A document containing R code (`apple.r`) is open in the top half of the screen. Below the divider, an R session is running, also inside Emacs. Code from the top pane is sent to the bottom with a keyboard shortcut, where it is evaluated by R. We can also jump down to the bottom pane and do work there. Small details like a lint checker, active line highlighting, and revision-control information are also visible.

kind of presentation software. You can use Microsoft PowerPoint or Apple's Keynote. Or, you can produce HTML or PDF slides directly from plain text documents.[4]

REPRODUCE WORK AND MINIMIZE ERROR

We have already seen how the right set of tools can save you time by automatically highlighting the syntax of your code, ensuring everything you cite ends up in your bibliography, picking out mistakes in syntax, and providing templates for commonly-used methods or functions. Your time is saved twice over: you don't repeat yourself, and you make fewer errors you'd otherwise have to fix. When it comes to managing ongoing projects, minimizing error means addressing two related problems. The first is to find ways to further reduce the opportunity for errors to creep in without you noticing. This is especially important when it comes to coding and analyzing data. The second is to find a way to figure out, retrospectively, what it was you did to generate a particular result. Using a revision control system gets us a good distance down this road. But there is more we can do at the level of particular reports or papers.

When you write code it is often in the process of doing some analysis on the fly. Ideas occur to you, you have a few things you want to look at, one thing leads to another. As a rule, you should try to document your work as you go. If you are writing an R script, then this usually means adding (brief, but useful) comments to your work to explain what it is a piece of code is meant to do. Is should also mean trying to write your code so that is readable. Code is like prose in this respect. Hadley Wickham's R Style Guide provides some useful guidelines about writing readable code. The R package lintr implements these principles—it acts like a copy-editor for your code. In Emacs you an use `lintr` automatically through a tool called flycheck.

You should also try not to repeat yourself when you write your code. A good rule is that if you find yourself copying and pasting chunks of code (for example, to draw the same sort of plot or run the same kind of model for a bunch of different variables) then you should pause and see if you can write a quick convenience function instead to automate things more effectively. That way, your code can be shorter and also less prone to the errors and inconsistencies that creep in through repeated copy-and-paste.

Errors in data analysis often well up out of the gap that typically exists between the procedure used to produce a figure or table in a paper and the subsequent use of that output later. In the ordinary way of doing things, you have the code for your data analysis in one file, the output it produced in another, and the text of your paper in a third file. You do the analysis, collect the output and copy the relevant results into your paper, often manually reformatting them on the way. Each of these transitions

[4] The actual business of *giving* talks based on your work is beyond the scope of this discussion. Suffice to say that there is plenty of good advice available via Google, and you should pay attention to it.

introduces the opportunity for error. In particular, it is easy for a table of results to get detached from the sequence of steps that produced it. Almost everyone who has written a quantitative paper has been confronted with the problem of reading an old draft containing results or figures that need to be revisited or reproduced (as a result of peer-review, say) but which lack any information about the circumstances of their creation. Academic papers take a long time to get through the cycle of writing, review, revision, and publication, even when you're working hard the whole time. It is not uncommon to have to return to something you did two years previously in order to answer some question or other from a reviewer. You do not want to have to do everything over from scratch in order to get the right answer. I am not exaggerating when I say that, whatever the challenges of replicating the results of someone else's quantitative analysis, after a fairly short period of time authors themselves find it hard to replicate their *own* work. Computer Science people have a term of art for the inevitable process of decay that overtakes a project simply in virtue of its being left alone on the hard drive for six months or more: bit–rot.

Use RMarkdown

An important way to caulk this gap is to use RMarkdown and knitr when doing quantitative analysis in R. We've already seen how to write plain-text documents in Markdown's lightweight format. RMarkdown allows you to incorporate code into this process. It is designed to integrate the plain-text documentation or writeup of a data analysis and its execution. You write the text of your paper (or, more often, your report documenting a data analysis) as normal. Whenever you want to run a model, produce a table or display a figure, rather than paste in the results of your work from elsewhere, you write down the R code that will produce the output you want. These "chunks" of code can be interspersed throughout the document. They are distinguished from the regular text by a special delimiter at the beginning and end of the block.

When you're ready, you `knit` the document (Xie 2015). That is, you feed your `.Rmd` file to R, which processes the code chunks, and produces a new `.md` where the code chunks have been replaced by their output. You can then turn that Markdown file into a PDF or HTML document. Relatedly, the `rmarkdown` library in R provides a `render()` function that takes you from `.Rmd` to HTML or PDF in a single step. This is what RStudio uses to produce your documents. Conversely, if you just want to extract the code you've written from the surrounding text, then you "tangle" the file, which results in an `.R` file. It's pretty straightforward in practice. The strength of this approach is that is makes it much easier to document your work properly. There is just one file for both the data analysis and the writeup. The output of the analysis is created on the fly, and the code to do it is embedded in the paper. If you need to do multiple but identical (or very similar) analyses of different bits of data, RMarkdown and `knitr` can make generating consistent and reliable reports much easier.

RMarkdown is one of several “literate programming” formats. The idea goes back to Donald Knuth, the pioneering theorist of computer science who developed the TeX typesetting system in his spare time. Although his focus was on documenting computer programs, in retrospect Knuth anticipated many of the main ideas—and developed several of the initial tools—for reproducible data analysis.

Figure 4, for instance, could be generated on the fly from source-code blocks included in the `.Rmd` source for this article. Sometimes we will want to only show the results produced by the code—in this case, Figure 4. But at other times we will want to display the code as well, as in Listing 2.

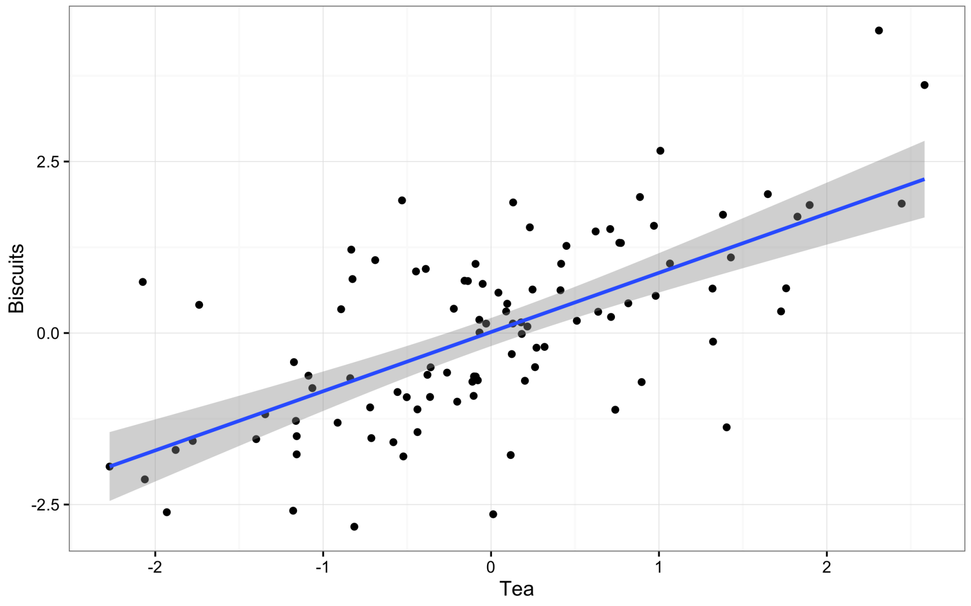


Figure 4: Tea and Biscuits

Listing 2: R code snippet.

```
library(ggplot2)
tea <- rnorm(100)
biscuits <- tea + rnorm(100,0,1.3)
data <- data.frame(tea, biscuits)
p <- ggplot(data, aes(x=tea, y=biscuits)) +
   geom_smooth(method="lm") +
   labs(x="Tea", y="Biscuits") + theme_bw()
print(p)
```

`knitr` and RMarkdown make it easy to produce HTML output, too. This makes for easy portability, conversion, and quick previewing while editing. You can work with RMarkdown files in any text editor, and Emacs has strong support for them. RStudio also comes with built-in support for `.Rmd` files and makes it very easy to produce HTML and PDF output, and to publish your reports to the web via its RPubs service.

The knitr website has numerous examples showing how it works. These range from the basic setup to more developed cases to more developed examples.

The literate programming approach has its limits. For large or complex analyses it can still make more sense to produce the final result in pieces rather than all at once in a single `.Rmd` file. This is one of the reasons it remains important to manage your projects using some kind of version control, so you can keep track of work that is needed but might not fit inside a single `.Rmd` document.

PULLING IT TOGETHER

We write papers. Those papers cite books and articles. They often incorporate tables and figures created in R. What we want to do is quickly turn a Markdown file containing things like that into a properly formatted scholarly paper, without giving up any of the necessary scholarly apparatus (on the output side) or the convenience and convertibilty of Markdown (on the input side). We want to easily get good-looking output from the same source in HTML, PDF, and DOCX formats. And we want to do that with an absolute minimum of—ideally, *no*—post-processing of the output beyond the basic conversion step. This is within our reach.

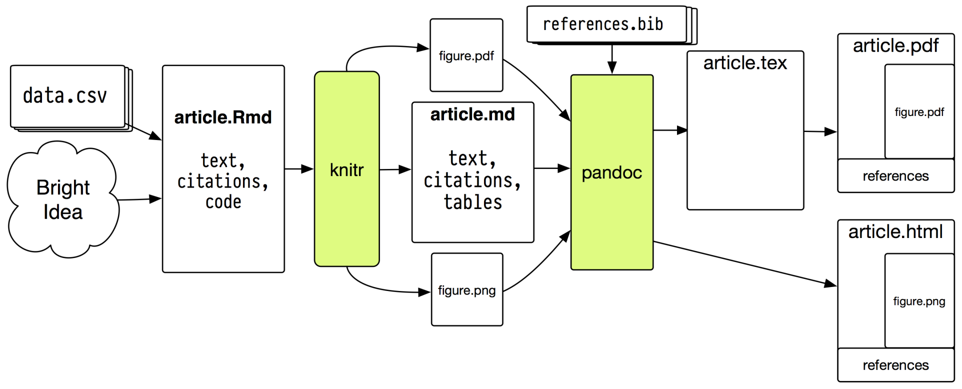


Figure 5: A plain-text document toolchain.

A sample document flow is shown in Figure 5. I promise it is less insane than it appears. Describing it all at once might make it sound a little crazy. But, at bottom, there are just two separable pieces. First, `knitr` converts `.Rmd` files to `.md` files. Second, John MacFarlane's superb Pandoc converts `.md` files to HTML, `.tex`, PDF, or Word formats. In both cases we use a few switches, templates and configuration files to do that nicely and with a minimum of fuss. You should install a standard set of Unix developer tools, which on OS X can conveniently be installed directly from the command line.[5] along with R, knitr, pandoc, and a TeX distribution. Note that the default set-ups for `knitr` and `pandoc`—the two key pieces of the process—will do most of what we want with no further tweaking. What I will you here are just the relevant options to use and switches to set for these tools, together with some templates and document samples showing how nice-looking output can be produced in practice.

I write everything in Emacs, but as I hope is clear by now, that doesn't matter. Use whatever text editor you like and just learn the hell out of it. The custom LaTeX style files were originally put together to let me write nice-looking `.tex` files directly, but now they just do their work in the background. Pandoc will use them when it converts things to PDF. The heavy lifting is done by the org-preamble-pdflatex.sty and memoir-article-styles files. If you install these files where LaTeX can find them—i.e., if you can compile a LaTeX document based on this example—then you are good to go. My BibTeX master file is also available, but you will probably want to use your own, changing references to it in the templates as appropriate. Second, we have the custom pandoc stuff. Here is the repository for that. Much of the material there is designed to go in the `~/.pandoc/` directory, which is where pandoc expects to find its configuration files. I have also set up a sample `md-starter` project and an `rmd-starter` project. These are the skeletons of project folders for a paper written in Markdown (just an `.md` file, with no R) and a paper beginning life as an `.Rmd` file. The sample projects contain the basic starter file and a `Makefile` to produce `.html`, `.tex`, `.pdf` and `.docx` files.

Listing 3: Markdown file with document metadata

```
---
title: "A␣Pandoc␣Markdown␣Article␣Starter"
author:
- name: Kieran Healy
  affiliation: Duke University
  email: kjhealy@soc.duke.edu
- name: Joe Bloggs
  affiliation: University of North Carolina, Chapel Hill
  email: joebloggs@unc.edu
date: January 2014
abstract: "Lorem␣ipsum␣dolor␣sit␣amet."
...
```

[5]Here's how. Open a terminal window and type `xcode-select --install`. You can install `pandoc` and many other tools using the Homebrew package manager.

```

# Introduction
Lorem ipsum dolor sit amet, consectetur adipisicing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna
aliqua [@fourcade13classsituat]. Notice that citation.

# Theory
Lorem ipsum dolor sit amet, consectetur adipisicing
elit, sed do eiusmod tempor incididunt ut labore et
dolore magna aliqua [@fourcade13classsituat].
```

Let's start with a straightforward markdown file—no R code yet, so nothing to the left of `article.md` line in Figure 5. The start of the sample `article-markdown.md` file is shown in Listing 3. The bit at the top is metadata, which pandoc understands. The HTML and LaTeX templates in the pandoc-templates repository are set up to use this metadata properly. Pandoc will take care of the citations directly. There is more than one way to have pandoc manage citations, but here we just use the most self-contained route. Simple documents can be contained in a single `.md` file. Documents including data analysis start life as `.Rmd` files which are then knitted into `.md` files and converted to PDF or HTML. At its simplest, a `mypaper.md` document can be converted to `mypaper.pdf` file by opening a terminal window and typing a command like the one in Listing 4.

Listing 4: The simplest way to convert a Markdown file to PDF with pandoc

```
pandoc mypaper.md -o mypaper.pdf
```

Because we will probably run commands like this a lot, it's convenient to automate them a little bit, and to add some extra bells and whistles to accommodate things we will routinely add to files, such as author information and other metadata, together with the ability to process bibliographic information and cross-references. These are handled by `pandoc` by turning on various switches in the command, and by ensuring a couple of external "filters" are present to manage the bibliographies and cross-references. Rather than type long commands out repeatedly, we will automate the process. This kind of automation is especially useful when our final output file might have a number of prerequisites before it can be properly produced, and we would like the computer to be a little bit smart about what needs to be re-processed and under what conditions. That way, for example, if a Figure has not changed we will not re-run the (possibly time-consuming) R script to create it again, unless we have to.

We manage this process using a tool called `make`. Inside our project folder we put a plain-text `Makefile` that contains some rules governing how to produce a *target* file that might have a number of *prerequisites*. In this case, a PDF or HTML file is the target, and the various figures and data tables are the prerequisites—if the code that produces the prerequisites changes, the final document will change too. `Make` starts from the final document and works backwards along the chain of prerequisites, re-compiling or re-creating them as needed. It's a powerful tool. For a good basic introduction, take a look at Karl Broman's "Minimal Make". (Incidentally, Karl Broman has a number of tutorials and guides providing accurate and concise tours of many of the tools and topics described here, including getting started with reproducible research, using git and GitHub, and working with knitr.)

Following Karl Broman's example, let's imagine that you have a paper, `paper.md` written in Markdown, that incorporates references to a figure, `fig1.pdf` generated by an R script, `fig1.r`. You could of course have an `.Rmd` file that produces the output, but there are situations where that isn't ideal. The end-point or target is the full article in PDF form. When the text of the paper changes in `paper.md`, then `paper.pdf` will have to be re-created. In the same way, when we change the content of `fig1.r` then `fig1.pdf` will need to be updated, and thus also `paper.pdf` will need to be re-created as well. Using `make` we can take care of the process.

Here is what a basic `Makefile` for our example would look like:

Listing 5: A simple Makefile

```
## Read as "mypaper.pdf depends on mypaper.md and fig1.pdf"
mypaper.pdf: mypaper.md fig1.pdf
    pandoc mypaper.md -o mypaper.pdf

## Read as "fig1.pdf depends on fig1.r"
fig1.pdf: fig1.r
    R CMD BATCH fig1.r
```

The big gotcha for `Makefiles` is that for no good reason they use the `<TAB>` key rather than spaces to indent the commands associated with rules. If you use spaces, `make` will not work. With the `Makefile` in Listing 5, typing `make` at the command line will have make check the state of the final target (`makefile.pdf`) and all its dependencies. If the target isn't present, `make` will create it in accordance with the rules specified. If the target *is* present, `make` will check to see if any of its prerequisites have changed since it was last created. If so, it will recreate the file. The chain of prerequisites propagates backwards, so that if you change `fig1.r`, then `make` will notice and re-run it to create `fig1.pdf` before running the commands in the rule to create `mypaper.pdf`. You can also choose to just invoke single rules from the makefile, e.g. by typing `make fig1.pdf` instead of `make` at the command line. This will evaluate just the `fig1.pdf` rule and any associated prerequisites.

For a simple example like this, make is mostly a minor convenience, saving you the trouble of typing a sequence of commands over and over to create your paper. However, it becomes very useful once you have projects with many documents and dependencies—for example, a dissertation consisting of separate chapters, each of which contains figures and tables, which in turn depend on various R scripts to set up and clean data. In those cases, `make` becomes a very powerful and helpful way of ensuring your final output really is up to date.

To deal with more complex projects and chains of prerequisites, `make` can make use of a number of variables to save you from (for example) typing out the name of every `figure-x.pdf` in your directory.

The `Makefile` in the sample `md-starter` project will convert any markdown files in the working directory to HTML, `.tex`, `.pdf`, or `.docx` files as requested. Typing `make html` at the command line will produce `.html` files from any `.md` files in the directory, for example. The PDF output (from `make pdf`) will look like this article, more or less. The different pieces of the `Makefile` define a few variables that specify the relationship between the different file types. In essence the rules say, for example, that all the PDF files in the directory depend on changes to an `.md` file with the same name; that the same is true of the HTML files in the directory, and so on. Then the show the `pandoc` commands that generate the output files from the markdown input. The Makefile itself is shown in Listing 6 makes use of a few variables as shorthand, as well as special variables like `$@` and `$<`, which mean "the name of the current target" and "the name of the current prerequisite", respectively.

Listing 6: A more complicated Makefile

```
## Extension (e.g. md, markdown, mdown).
## for all markdown files in the directory
MEXT = md

## Variable expands to a list of all markdown files
## in the working directory
SRC = $(wildcard *.$(MEXT))

## Location of Pandoc support files.
PREFIX = /Users/kjhealy/.pandoc

## Location of your working bibliography file
BIB = /Users/kjhealy/Documents/bibs/socbib-pandoc.bib

## CSL stylesheet (located in the csl folder of the PREFIX directory).
CSL = apsa

## x.pdf depends on x.md, x.html depends on x.md, etc
PDFS=$(SRC:.md=.pdf)
HTML=$(SRC:.md=.html)
TEX=$(SRC:.md=.tex)
DOCX=$(SRC:.md=.docx)
```

```
## Rules -- make all, make pdf, make html. The ‘clean‘ rule is below.
all:    $(PDFS) $(HTML) $(TEX) $(DOCX)
pdf:    clean $(PDFS)
html:   clean $(HTML)
tex:    clean $(TEX)
docx:   clean $(DOCX)

## The commands associated with each rule.
## This first one is run when ‘make html‘ is typed.
## Read the rule as "When making the html file,
## run this pandoc command if the .md file has changed."
%.html: %.md
    pandoc -r markdown+simple_tables+table_captions+yaml_metadata_block \
    -w html -S --template=$(PREFIX)/templates/html.template \
    --css=$(PREFIX)/marked/kultiad-serif.css --filter pandoc-crossref \
    --filter pandoc-citeproc --csl=$(PREFIX)/csl/$(CSL).csl \
    --bibliography=$(BIB) -o $@ $<

## Same goes for the other file types. Watch out for the TAB before ’pandoc’
%.tex:  %.md
    pandoc -r markdown+simple_tables+table_captions+yaml_metadata_block \
    --listings -w latex -s -S --latex-engine=pdflatex \
    --template=$(PREFIX)/templates/latex.template \
    --filter pandoc-crossref --filter pandoc-citeproc \
    --csl=$(PREFIX)/csl/ajps.csl --filter pandoc-citeproc-preamble \
    --bibliography=$(BIB) -o $@ $<

%.pdf:  %.md
    pandoc -r markdown+simple_tables+table_captions+yaml_metadata_block \
    --listings -s -S --latex-engine=pdflatex \
    --template=$(PREFIX)/templates/latex.template \
    --filter pandoc-crossref --filter pandoc-citeproc \
    --csl=$(PREFIX)/csl/$(CSL).csl --filter pandoc-citeproc-preamble \
    --bibliography=$(BIB) -o $@ $<

%.docx: %.md
    pandoc -r markdown+simple_tables+table_captions+yaml_metadata_block \
    -s -S --filter pandoc-crossref --csl=$(PREFIX)/csl/$(CSL).csl \
    --bibliography=$(BIB) -o $@ $<

clean:
    rm -f *.html *.pdf *.tex *.aux *.log *.docx

.PHONY: clean
```

Note that the `pandoc` commands are interpreted single lines of text, not several lines separated by the `<return>` key. But you can use the `\` symbol to tell `make` to continue to the next line without a break. With this Makefile, typing `make pdf` would take all the `.md` files in the directory one at a time and run the pandoc command to turn each one into a PDF, using the APSR reference style, my latex template, and a `.bib` file called `socbib-pandoc.bib`.

You shouldn't use this `Makefile` blindly. Take the time to learn how `make` works and how it can help your project. The official manual is pretty clear. Make's backward-looking chain of prerequisites can make it tricky to write rules for complex projects. When writing or inspecting a `Makefile` and its specific rules, it can be helpful to use the `--dry-run` switch, as in `make --dry-run`. This will print out the sequence of commands `make` would run, but without actually executing them. You can try this with the `Makefile` in Listing 6 in a directory with at least one `.md` file in it. For example, look at the list of commands produced by `make pdf --dry-run` or `make docx --dry-run` or `make clean --dry-run`.

The particular steps needed for many projects may be quite simple, and not require the use of any variables or other frills. If you find yourself repeatedly running the same sequence of commands to assemble a document (e.g. cleaning data; running preliminary code; producing figures; assembling a final document) then `make` can do a lot to automate the process. For further examples of `Makefiles` doing things relevant to data analysis, see Lincoln Mullen's discussion of the things he uses `make` to manage.

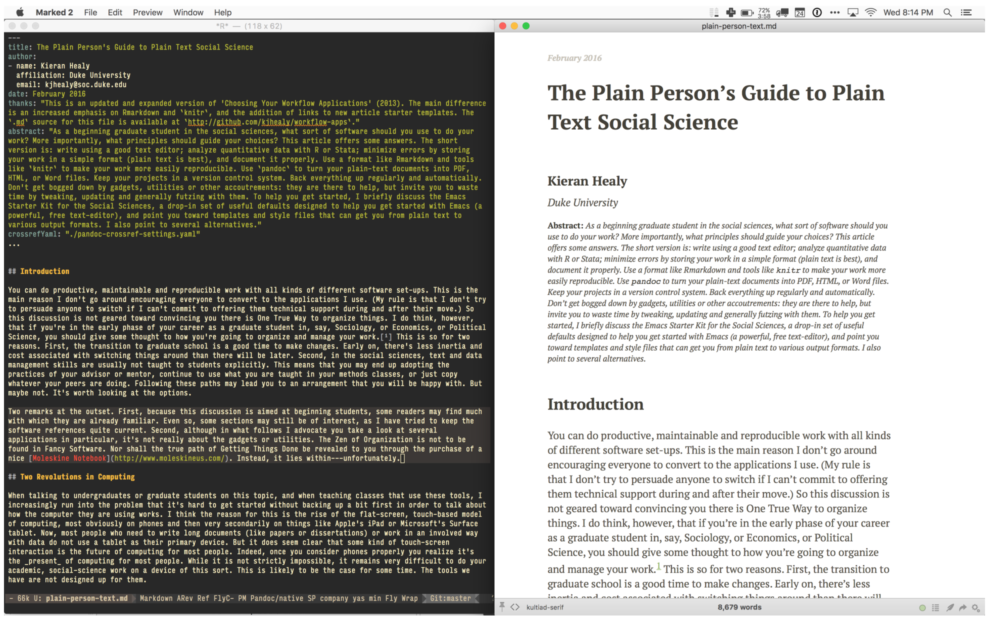
The particular steps needed for many projects may be quite simple, and not require the use of any variables or other frills. If you find yourself repeatedly running the same sequence of commands to assemble a document (e.g. cleaning data; running preliminary code; producing figures; assembling a final document) then `make` can do a lot to automate the process.

The examples directory includes a sample `.Rmd` file. The code chunks in the file provide examples of how to generate tables and figures in the document. In particular they show some useful options that can be passed to knitr. Consult the `knitr` project page for extensive documentation and many more examples. To produce output from the `article-knitr.Rmd` file, you could of course launch R in the working directory, load `knitr`, and process the file. This produces the `article-knitr.md` file, together with some graphics in the `figures/` folder (and some working files in the `cache/` folder). We set things up in the `.Rmd` file so that `knitr` produces both PNG and PDF versions of whatever figures are generated by R. That prepares the way for easy conversion to HTML and LaTeX. Once the `article-knitr.md` file is produced, HTML, `.tex`, and PDF versions of it can be produced as before, by typing `make` at the command line. But of course there's no reason `make` can't automate that first step, too. The `rmd-starter` project has a sample `Makefile` that begins with the `.Rmd` files in the directory and produces the outputs from there.

Using Marked

In everyday use, I find Brett Terpstra's application Marked to be a very useful way of previewing text while writing. Marked shows you your Markdown files as HTML, updating the preview on the fly whenever changes are saved in the Markdown file. It can render ordinary Markdown by default, but it also supports `pandoc` as a custom processor. This means it can manage the various extra bells and whistles of scholarly formatting discussed so far. Essentially, you tell Marked run a `pandoc` command similar or identical the one shown above to generate its HTML previews. You do this in the "Advanced" tab of Marked's preferences. The "Path" field in the preferences dialog contains the full path to `pandoc`, and the "Args" field contains all the relevant command switches—in my case, the same as in the Makefile above.

When editing your markdown file in your favorite text editor, you point Marked at the file and get a live preview. You can add the CSS files in the pandoc-templates repository to the list of Custom CSS files Marked knows about, via the "Style" tab in the Preferences window. That way, Marked's preview will look the same as the HTML file that's produced. Figure 6 shows what this looks like in practice.



```
---
title: The Plain Person's Guide to Plain Text Social Science
author:
- name: Kieran Healy
  affiliation: Duke University
  email: kjhealy@soc.duke.edu
date: February 2016
thanks: "This is an updated and expanded version of 'Choosing Your Workflow Applications' (2013). The main difference
is an increased emphasis on Rmarkdown and `knitr`, and the addition of links to new article starter templates. The
`.md` source for this file is available at `http://github.com/kjhealy/workflow-apps`."
abstract: "As a beginning graduate student in the social sciences, what sort of software should you use to do your
work? More importantly, what principles should guide your choices? This article offers some answers. The short
version is: write using a good text editor; analyze quantitative data with R or Stata; minimize errors by storing
your work in a simple format (plain text is best), and document it properly. Use a format like Rmarkdown and tools
like `knitr` to make your work more easily reproducible. Use `pandoc` to turn your plain-text documents into PDF,
HTML, or Word files. Keep your projects in a version control system. Back everything up regularly and automatically.
Don't get bogged down by gadgets, utilities or other accoutrements: they are there to help, but invite you to waste
time by tweaking, updating and generally futzing with them. To help you get started, I briefly discuss the Emacs
Starter Kit for the Social Sciences, a drop-in set of useful defaults designed to help you get started with Emacs (a
powerful, free text-editor), and point you toward templates and style files that can get you from plain text to
various output formats. I also point to several alternatives."
crossrefYaml: "./pandoc-crossref-settings.yaml"
...


## Introduction

You can do productive, maintainable and reproducible work with all kinds of different software set-ups. This is the
main reason I don't go around encouraging everyone to convert to the applications I use. (My rule is that I don't try
to persuade anyone to switch if I can't commit to offering them technical support during and after their move.) So
this discussion is not geared toward convincing you there is One True Way to organize things. I do think, however,
that if you're in the early phase of your career as a graduate student in, say, Sociology, or Economics, or Political
Science, you should give some thought to how you're going to organize and manage your work.[^1] This is so for two
reasons. First, the transition to graduate school is a good time to make changes. Early on, there's less inertia and
cost associated with switching things around than there will be later. Second, in the social sciences, text and data
management skills are usually not taught to students explicitly. This means that you may end up adopting the
practices of your advisor or mentor, continue to use what you are taught in your methods classes, or just copy
whatever your peers are doing. Following these paths may lead you to an arrangement that you will be happy with. But
maybe not. It's worth looking at the options.

Two remarks at the outset. First, because this discussion is aimed at beginning students, some readers may find much
with which they are already familiar. Even so, some sections may still be of interest, as I have tried to keep the
software references quite current. Second, although in what follows I advocate you take a look at several
applications in particular, it's not really about the gadgets or utilities. The Zen of Organization is not to be
found in Fancy Software. Nor shall the true path of Getting Things Done be revealed to you through the purchase of a
nice [Moleskine Notebook](http://www.moleskineus.com/). Instead, it lies within---unfortunately.

## Two Revolutions in Computing

When talking to undergraduates or graduate students on this topic, and when teaching classes that use these tools, I
increasingly run into the problem that it's hard to get started without backing up a bit first in order to talk about
how the computer they are using works. I think the reason for this is the rise of the flat-screen, touch-based model
of computing, most obviously on phones and then very secondarily on things like Apple's iPad or Microsoft's Surface
tablet. Now, most people who need to write long documents (like papers or dissertations) or work in an involved way
with data do not use a tablet as their primary device. But it does seem clear that some kind of touch-screen
interaction is the future of computing for most people. Indeed, once you consider phones properly you realize it's
the _present_ of computing for most people. While it is not strictly impossible, it remains very difficult to do your
academic, social-science work on a device of this sort. This is likely to be the case for some time. The tools we
have are not designed up for them.

- 66k U: plain-person-text.md  Markdown ARev Ref FlyC- PM Pandoc/native SP company yas min Fly Wrap  Git:master
```

February 2016

The Plain Person's Guide to Plain Text Social Science

Kieran Healy

Duke University

Abstract: *As a beginning graduate student in the social sciences, what sort of software should you use to do your work? More importantly, what principles should guide your choices? This article offers some answers. The short version is: write using a good text editor; analyze quantitative data with R or Stata; minimize errors by storing your work in a simple format (plain text is best), and document it properly. Use a format like Rmarkdown and tools like* `knitr` *to make your work more easily reproducible. Use* `pandoc` *to turn your plain-text documents into PDF, HTML, or Word files. Keep your projects in a version control system. Back everything up regularly and automatically. Don't get bogged down by gadgets, utilities or other accoutrements: they are there to help, but invite you to waste time by tweaking, updating and generally futzing with them. To help you get started, I briefly discuss the Emacs Starter Kit for the Social Sciences, a drop-in set of useful defaults designed to help you get started with Emacs (a powerful, free text-editor), and point you toward templates and style files that can get you from plain text to various output formats. I also point to several alternatives.*

Introduction

You can do productive, maintainable and reproducible work with all kinds of different software set-ups. This is the main reason I don't go around encouraging everyone to convert to the applications I use. (My rule is that I don't try to persuade anyone to switch if I can't commit to offering them technical support during and after their move.) So this discussion is not geared toward convincing you there is One True Way to organize things. I do think, however, that if you're in the early phase of your career as a graduate student in, say, Sociology, or Economics, or Political Science, you should give some thought to how you're going to organize and manage your work.[1] This is so for two reasons. First, the transition to graduate school is a good time to make changes. Early on, there's less

Figure 6: Working on this document in Emacs (left), with a live HTML version displayed in Marked (right).

The upshot of all of this is powerful editing using Emacs, ESS, R, and other tools; flexible conversion using pandoc; quick and easy previewing via HTML and Marked;

and high-quality PDF typesetting at the same time (or whenever needed). All of it can be generated directly from plain text, and will include almost all of the machinery most scholarly papers need, most notably properly labeled Tables and Figures that can be cross-referenced in the text. While it all may seem quite complex when laid out in this way, in use it is quite straightforward, and doesn't require any thought when in use. I just live in my text editor. The various scripts and settings do their work quietly, and I get the formatted output I want.

An Emacs Starter Kit

A step-by-step guide to downloading and installing every piece of software I've mentioned so far is beyond the scope of this discussion. But let's say you take the plunge and download Emacs, a TeX distribution, R, and maybe even Git. Now what? If you're going to work in Emacs, there are a variety of tweaks and add-ons that are very helpful but not set by default. To make things a little easier, I maintain an Emacs Starter Kit for the Social Sciences. It's designed to smooth out Emacs' rough edges by giving you a drop-in collection of default settings, as well as automatically installing some important add-on packages. It will, I hope, help you skirt the abyss of Setting Things Up Forever.

DO I HAVE TO USE ALL THIS STUFF?

> Installation/setup/whatever is always harder and much more poorly documented than mere usage — Jenny's Law

Pros and Cons

Running your data analysis in R, writing documents in Markdown or RMarkdown, doing both inside Emacs, processing them with `pandoc`, tracking things with Git and using (behind the scenes) various Unix tools and LaTeX … this all sounds rather complicated. It has four main advantages. First, these formats, tools, and applications are all free. You can try them out without much in the way of monetary expense. (Your time may be a different matter, but although you don't believe me, you have more of that now than you will later.) Second, they are all open-source projects and are all available for Mac OS X, Linux and Windows. Portability is important, as is the long-term viability of the platform you choose to work with. If you change your computing system, your work can move with you easily. Third, they allow you to do your work in a portable, documented, and reproducible way. And fourth, the applications are closely integrated. Everything (including version control) can work inside Emacs. All of them can work directly with or take advantage of the others.

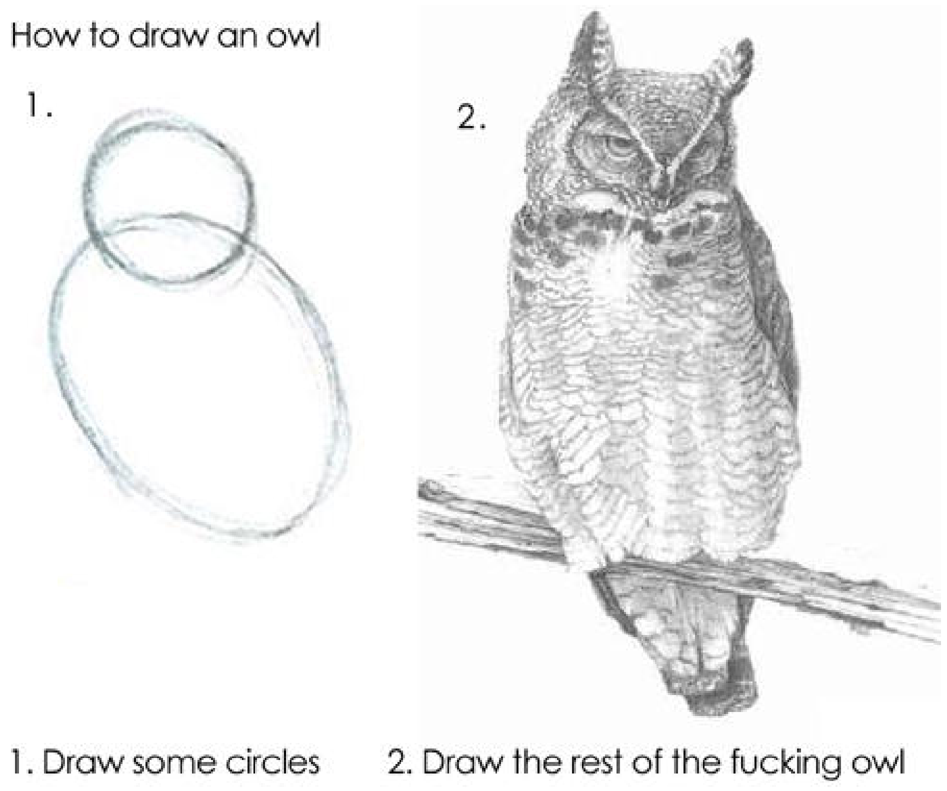


Figure 7: How to Draw an Owl

None of these tools is perfect. They can do very useful and important things for you, but they are not magic. There are bad habits associated with working in plain text, just as there are bad habits associated with writing everything in word. These tools are powerful, but they can be tedious to learn. However, you don't have to start out using all of them at once, or learn everything about them right away—the only thing you really, *really* need to start doing *immediately* is keeping good backups. There are a number of ways to try these tools out in whole or in part. You could try writing something in Markdown first, using any text editor. You could begin using R with RStudio. Revision control is more beneficial when implemented at the beginning of projects, and best of all when committing changes to a project becomes a habit of work. But it can be added at any time.

You are not condemned to use these applications forever, either. RMarkdown and (especially) Markdown documents can be converted into many other formats. Your text files are editable in any other text editor and on any other computer. Statistical code is by nature much less portable, but the openness of R means that it is not likely to become obsolete or inaccessible any time soon. In everyday use, you may find that documents start life as plain-text, Markdown-formatted notes jotted down on your phone, or computer; then they become longer `.md` files that grow references and figures and so on; and eventually migrate to Word or Google Docs or something similar if you acquire a collaborator or the time comes to submit a manuscript to a journal.

A disadvantage of these particular applications is that I'm in a minority with respect to other people in my field. This is less and less true in the case of R, and more recently with tools like Git as well. Writing papers in RMarkdown or Markdown is less common. Most people use Microsoft Word to write papers, and if you're collaborating with people (people you can't boss around, I mean) this can be an issue. It is usually easier to use applications like Word than convert people to a plain-text workflow. If you do, at least try and implement some of the principles discussed here when it comes to tracking changes to documents and managing the code and output of your data analysis.

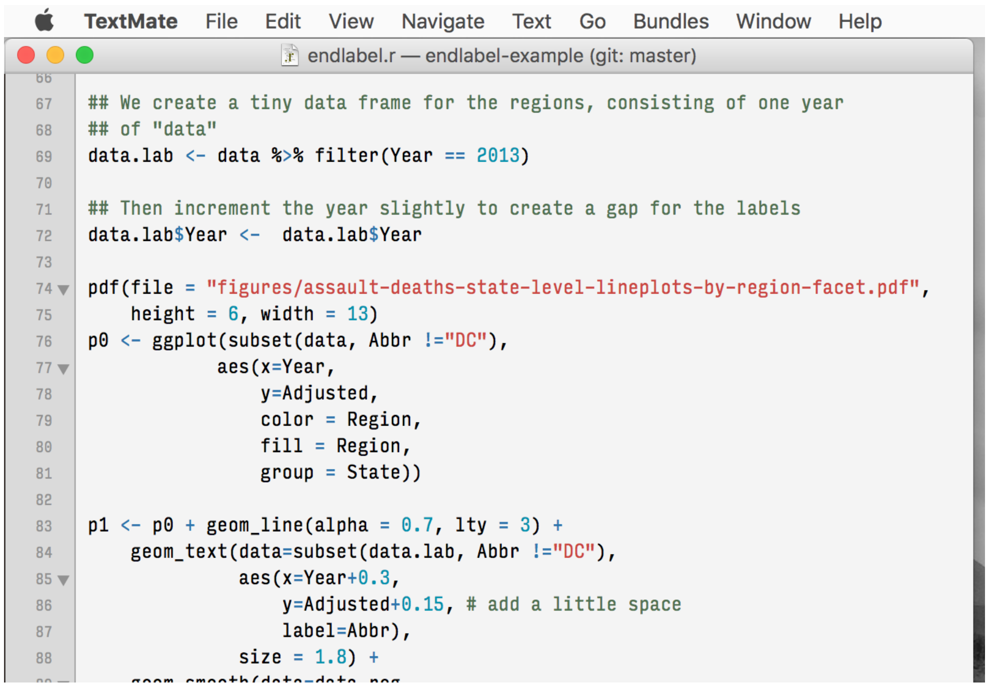
Alternatives Might Be Better

There are many other applications you might put at the center of your workflow, depending on need, personal preference, willingness to pay some money, or desire to work on a specific platform. For text editing, especially, there is a plethora of choices. On the Mac, quality editors include BBEdit (beloved of many web developers, but with relatively little support for R beyond syntax highlighting), and TextMate 2 (shown in Figure 8). On Linux, the standard alternative to Emacs is vi or Vim, but there are many others. For Windows there is Textpad, WinEdt, UltraEdit, and NotePad++ amongst many others. Most of these applications have strong support for LaTeX and some also have good support for statistics programming.

Sublime Text 3 is a cross-platform text editor under active development, and with an increasingly large user base. Sublime Text is fast, fluid, and contains a powerful plugin system based on the Python programming language. Uniquely amongst alternatives to Emacs and ESS, Sublime Text includes a well-developed REPL that allows R to be run easily inside the editor.[6] Sublime Text costs $70.

Finally, and as noted throughout this article, for a different approach to working with R you should consider RStudio. A screenshot is shown in Figure 9. Although it appears quite late in this discussion, it might well be your first choice. I use it when teaching. It is not a text editor but rather an "IDE", an integrated development environment. Your code and figures, together with an R console, documentation, and other output are all displayed in different panes and tabs of RStudio's application

[6]TextMate also has some support for this way of working, but it is conceived and accomplished a little differently.

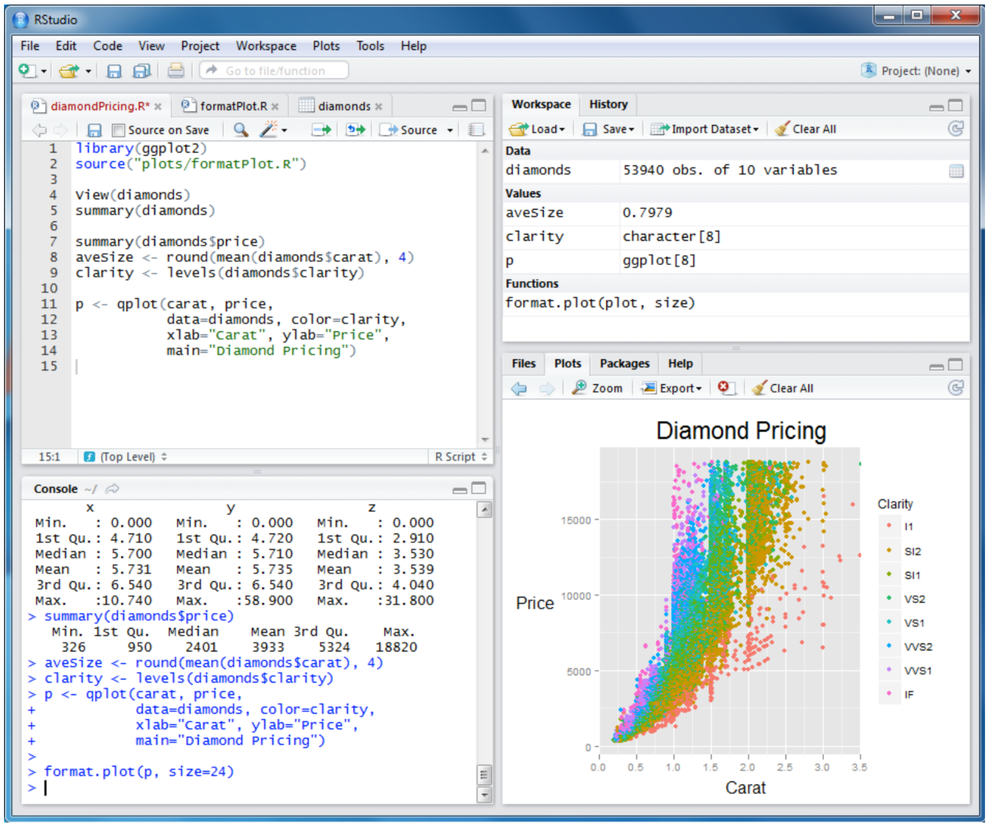


```r
## We create a tiny data frame for the regions, consisting of one year
## of "data"
data.lab <- data %>% filter(Year == 2013)

## Then increment the year slightly to create a gap for the labels
data.lab$Year <-  data.lab$Year

pdf(file = "figures/assault-deaths-state-level-lineplots-by-region-facet.pdf",
    height = 6, width = 13)
p0 <- ggplot(subset(data, Abbr !="DC"),
            aes(x=Year,
                y=Adjusted,
                color = Region,
                fill = Region,
                group = State))

p1 <- p0 + geom_line(alpha = 0.7, lty = 3) +
    geom_text(data=subset(data.lab, Abbr !="DC"),
              aes(x=Year+0.3,
                  y=Adjusted+0.15, # add a little space
                  label=Abbr),
              size = 1.8) +
```

Figure 8: Part of an R file being edited in TextMate.

window. Data and script files are managed via various windows and menus. RStudio is available for Mac OS X, Windows, and Linux. It intergrates nicely with R's help files. It understands `knitr` and Git. As discussed above, it has full support for Rmarkdown and generates HTML, PDF, and other formats for you very easily. It is the easiest way by far to get into using R, and provides a straightforward way to manage many of the tools already discussed here.

For statistical analysis in the social sciences, the main alternative to R is Stata. Stata is not free, but like R it is versatile, powerful, extensible and available for all the main computing platforms. It has a large body of user-contributed software. In recent versions its graphics capabilities have improved a great deal, as has its editor. ESS can highlight Stata `.do` files in the same way as it can do for R. Other editors can also be made to work with Stata. More recently, Python has been coming into frequent use in the social sciences. Python is a general-purpose computing language that is relatively straightforward to learn. It is often used for manipulating, cutting, and cleaning data prior to analysis in applications like R or Stata. But it is also increasingly a scientific computing platform in its own right. SciPy is a useful place to begin learning about Python's capabilities on this front. Like R and RMarkdown, it has good support for



```
library(ggplot2)
source("plots/formatPlot.R")

View(diamonds)
summary(diamonds)

summary(diamonds$price)
aveSize <- round(mean(diamonds$carat), 4)
clarity <- levels(diamonds$clarity)

p <- qplot(carat, price,
           data=diamonds, color=clarity,
           xlab="Carat", ylab="Price",
           main="Diamond Pricing")
```

```
       x                y                z
 Min.   : 0.000   Min.   : 0.000   Min.   : 0.000
 1st Qu.: 4.710   1st Qu.: 4.720   1st Qu.: 2.910
 Median : 5.700   Median : 5.710   Median : 3.530
 Mean   : 5.731   Mean   : 5.735   Mean   : 3.539
 3rd Qu.: 6.540   3rd Qu.: 6.540   3rd Qu.: 4.040
 Max.   :10.740   Max.   :58.900   Max.   :31.800
> summary(diamonds$price)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    326     950    2401    3933    5324   18820
> aveSize <- round(mean(diamonds$carat), 4)
> clarity <- levels(diamonds$clarity)
> p <- qplot(carat, price,
+            data=diamonds, color=clarity,
+            xlab="Carat", ylab="Price",
+            main="Diamond Pricing")
>
> format.plot(p, size=24)
> 
```

Data	
diamonds	53940 obs. of 10 variables
Values	
aveSize	0.7979
clarity	character[8]
p	ggplot[8]
Functions	
format.plot(plot, size)	

Figure 9: RStudio running on Windows.

literate programming through tools like iPython Notebooks. Naturally, Emacs has good support for working with Python.

Amongst social scientists, revision control is perhaps the least widely-used of the tools I have discussed. But I am convinced that it is the most important one over the long term. While tools like Git take a little getting used to both conceptually and in practice, the services they provide are extremely useful. It is already quite easy to use version control in conjunction with most of the text editors discussed above. There are also full-featured clients like Tower that allow you to administer git without having to use the command line. Taking a longer view, version control is likely to become more widely available through intermediary services or even as part of the basic functionality of operating systems.

A BROADER PERSPECTIVE

It would be nice if all you needed to do your work was a box software of software tricks and shortcuts. But of course it's a bit more complicated than that. In order to get to the point where you can write a paper, you need to be organized enough to have read the right literature, maybe collected some data, and most importantly asked an interesting question in the first place. No amount of software is going to solve those problems for you. Too much concern with the details of your setup hinders your work. Indeed—and I speak from experience here—this concern is itself a kind self-imposed distraction that placates work-related anxiety in the short term while storing up more of it for later.[7] On the hardware side, there's the absurd productivity counterpart to the hedonic treadmill, where for some reason it's hard to get through the to-do list even though the cafe you're at contains more computing power than was available to the Pentagon in 1965. On the software side, the besetting vice of software is the tendency to waste a lot of your time installing, updating, and generally obsessing about it.[8] Even more generally, efficient workflow habits are themselves just a means to the end of completing the projects you are really interested in, of making the things you want to make, of finding the answers to the questions that brought you to graduate school. The process of idea generation and project management can be run well, too, and perhaps even the business of choosing what the projects should be in the first place. But this is not the place—and I am not the person—to be giving advice about that.

All of which is just to reiterate two things. First, I am not advocating these tools on the grounds that they will make you more "productive". Rather, they may help you stay in control of—and able to reproduce—your own prior work. That is an important difference. If you care about getting the right answer in your data analysis, or at least being able to repeatedly get the same probably wrong answer, then tools that enhance this sort of control should appeal to you. Second, even with that caveat it is still the *principles* of workflow management that are important. The software is just a means to an end. One of the smartest, most productive people I've ever known spent half of his career writing on a typewriter and the other half on an ancient IBM Displaywriter. His backup solution for having hopelessly outdated hardware was to keep a spare Displaywriter in a nearby closet, in case the first one broke. It never did.

[7] See Merlin Mann, amongst others, for more on this point.

[8] Mike Hall's brilliant "Org-Mode in your Pocket is a GNU-Shaped Devil" makes this point very well.

APPENDIX: LINKS TO RESOURCES

Basic Tools

- Apple's Developer Tools Unix toolchain. Install directly with `xcode-select --install`, or just try to use e.g. `git` from the terminal and have OS X prompt you to install the tools.
- Homebrew package manager. A convenient way to install several of the tools here, including Emacs and Pandoc.
- Emacs. A powerful text editor. Ready-to-go Mac version at Emacs for Mac OS X.
- R. A platform for statistical computing.
- knitr. Reproducible plain-text documents from within R.
- Python and SciPy. Python is a general-purpose programming language increasingly used in data manipulation and analysis.
- RStudio. An IDE for R. The most straightforward way to get into using R and RMarkdown.
- TeX and LaTeX. A typesetting and document preparation system. You can write files in `.tex` format directly, or you can just have it available in the background for other tools to use. The MacTeX Distribution is the one to install for OS X.
- Pandoc. Converts plain-text documents to and from a wide variety of formats. Can be installed with Homebrew. Be sure to also install `pandoc-citeproc` for processing citations and bibliographies, and `pandoc-crossref` for producing cross-references and labels.
- Git. Version control system. Installs with Apple's Developer Tools, or get the latest version via Homebrew.
- GNU Make. You tell `make` what the steps are to create the pieces of a document or program. As you edit and change the various pieces, it automatically figures out which pieces need to be updated and recompiled, and issues the commands to do that. See Karl Broman's Minimal Make for a short introduction. Make will be installed automatically with Apple's developer tools.
- lintr and flycheck. Tools that nudge you to write neater code.

Helpers and Templates

- Emacs Starter Kit for the Social Sciences. Set Emacs up to use many of the tools described in this guide.
- Pandoc Templates. LaTeX and HTML templates, together with Pandoc configuration files and other things needed to produce good-looking PDF, HTML, and Word documents from plain text sources using Pandoc.
- `md-starter` project and `rmd-starter` project. Assuming you have the tools and

Pandoc/LaTeX templates installed, these skeleton project folders contain a basic `.md` or `.rmd` starter file and a `Makefile` to produce `.html`, `.tex`, `.pdf` and `.docx` files as described in this guide.
- RMarkdown Cheatsheet An overview of Markdown and RMarkdown conventions.
- RStudio Cheatsheets Other quick guides, including a more comprehensive RMarkdown reference and a information about using RStudio's IDE, and some of the main tools in R.

Guides

- R Style Guide. Write readable code.
- knitr Documentation and examples for `knitr` by its author, Yihui Xie. There is also a knitr book covering the same ground in more detail.
- Rmarkdown documentation from the makers of RStudio. Lots of good examples.
- Plain Person's Guide The git repository for this project.
- Jenny Bryan's Stat 545. Notes and tutorials for a Data Analysis course taught by Jennifer Bryan at the University of British Columbia. Lots of useful material.
- Karl Broman's Tutorials and Guides Accurate and concise guides to many of the tools and topics described here, including getting started with reproducible research, using git and GitHub, and working with knitr.
- Makefiles for OCR and converting Shapefiles. Some further examples of `Makefiles` in the data-analysis pipeline, by Lincoln Mullen

Paid Applications and Services

- Backblaze. Secure off-site backup.
- Crashplan. Secure off-site backup.
- GitHub. Host public Git repositories for free. Pay to host private ones. Also a source for publicly available code (e.g. R packages and utilities) written by other people.
- Marked 2. Live HTML previewing of Markdown documents. Mac OS X only.
- Sublime Text. Python-based text editor.
- Zotero, Mendeley, and Papers are citation managers that incorporate PDF storage, annotation and other features. Zotero is free to use. Mendeley has a premium tier. Papers is a paid application after a trial period. I don't use these tools much, but that's not for any strong principled reason—mostly just intertia. If you use one and want to integrate with the material here, just make sure it can export to BibTeX/BibLaTeX files. Papers, which I've used most recently, can handily output citation keys in pandoc's format amongst several others.

REFERENCES

Dalgaard, Peter. 2008. *Introductory Statistics with R*. Second edition. New York: Springer.

Fox, John. 2002. *An R and S-Plus Companion to Applied Regression*. Thousand Oaks: Sage.

Gelman, Andrew, and Jennifer Hill. 2007. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. New York: Cambridge University Press.

Harrell, Frank. 2016. *Regression Modeling Strategies*. Second. New York: Springer.

Healy, Kieran, and James Moody. 2014. "Data Visualization in Sociology." *Annual Review of Sociology* 40: 105–28.

Maindonald, John, and John Braun. 2003. *Data Analysis and Graphics Using R: An Example-Based Approach*. New York: Cambridge University Press.

Matloff, Norman. 2011. *The Art of R Programming*. San Francisco: No Starch Press.

Venables, W.N., and B.D. Ripley. 2002. *Modern Applied Statistics with S*. Fourth. New York: Springer.

Xie, Yihui. 2015. *Dynamic Documents with R and Knitr*. Second. New York: Chapman; Hall.