Computational Sociology (PGSP11583)

Christopher Barrie

2022-08-23T00:00:00+03:00

Table of contents

Computational Sociology

This is the course book we will be using for Computational Sociology (PGSP11583).

```
print("Computational Sociology")
```

```
[1] "Computational Sociology"
```

> The book is a "live" document meaning I will be updating as we progress together through the course.

License

Course Overview

This below gives details of course structure, outcomes, and assessment.

Learning outcomes

This course will give students training in the use of various computational methods in the social sciences. The course will prepare students for dissertation work that uses digital trace data and/or computational methods and will provide hands-on training in the use of the R programming language and (some) Python.

The course will provide a venue for seminar discussion of examples using these methods in the empirical social sciences as well as lectures on the technical and/or statistical dimensions of their application.

Course structure

We will be using this online book for the ten-week course in "Computational Sociology" (PGSP11583). Each chapter contains the readings for that week. The book also includes worksheets with example code for how to conduct some of the text analysis techniques we discuss each week.

Each week (with the partial exception of week 1), we will be discussing, alternately, the substantive and technical dimensions of published research in the empirical social sciences.

Course theme

In order to discipline the course, I have decided to focus on one theme that is currently making headlines: the political consequences of social media.

With this in mind, every two weeks we will be studying a different phenomenon that has gained media attention: e.g., echo chambers, misinformation, violence.

On alternate weeks, we will discuss how these phenomena have been studied—using both computational and non-computational methods. In the subsequent week, we will then go through some of the technical dimensions of the methods used in the papers we study.

Course pre-preparation

NOTE: Before the lecture in Week 2, students should complete two introductory R exercises.

1. First, you should consult the introduction to file directories and Github here. This is crucial to properly logging and saving the work you produce.
2. Second, you should consult the worksheet here, which is an introduction to setting up and understanding the very basics of working in R. S
3. Third, Ugur Ozdemir has provided a more comprehensive introductory R course for the Research Training Centre at the University of Edinburgh and you can follow the instructions here to access this.

Reference sources

There are two main reference texts that will be of use during this course:

- Wickham, Hadley and Garrett Grolemund. R for Data Science: https://r4ds.had.co.nz/
- Salganik, Matt. Bit by Bit: Social Research in the Digital Age: https://www.bitbybitbook.com/

Assessment

Fortnightly worksheets

Each fortnight, I will provide you with one worksheet that walks you through how to implement a different computational technique. At the end of these worksheets you will find a set of questions. **You should buddy up with someone else in your class and go through these together.**

This is called "pair programming" and there's a reason we do this. Firstly, coding can be an isolating and difficult thing—it's good to bring a friend along for the ride! Secondly, if there's something you don't know, maybe your buddy will. This saves you both time. Thirdly, your buddy can check your code as you write it, and vice versa. Again, this means both of you are working together to produce and check something as you go along.

At the subsequent week's lecture, I will pick on a pair at random to answer each one of that worksheet's questions (i.e., there is ~1/3 chance you're going to get picked each week). I will ask you to walk us through your code. And remember: it's also fine if you struggled and didn't get to the end! If you encountered an obstacle, we can work through that together. All that matters to me is that you **try**.

Fortnightly flash talks

On the weeks where you are not going to be tasked with a coding assignment, you're not off the hook… I will again be selecting a pair at random (the same as your coding pair) to talk me through one of the readings. I will pick a different pair for each reading (i.e., ~ 1/3 chance again).

Don't let this be cause of great anguish: I just want **two or three minutes** where you lay out for me: 1) the research question; 2) the data source; 3) the method; 4) the findings; 5) what you thought its limitations were. The main portion of your flash talk should focus on element 5).

For this last one, you will want to think about 1)-4); i.e., you will want to think about whether it really answered the research question, whether the data was appropriate for answering that question, whether the method was appropriate for answering that question, and whether the results show what the author claims they show. **I will provide you wish an example flash talk at the first lecture.**

Final assessment

Assessment takes the form of **one** summative assessment. This will be a 4000 word essay on a subject of your choosing (with prior approval by me). For this, you will be required to select from a range of data sources I will provide. You may also suggest your own data source.

You will be asked to: a) formulate a research question; b) use at least one computational technique that we have studied; c) conduct an analysis of the data source you have provided; d) write up the initial findings; and e) outline potential extensions of your analysis.

You will then provide the code you used in reproducible (markdown) format and will be assessed on both the substantive content of your essay contribution (the social science part) as well as your demonstrated competency in coding and text analysis (the computational part).

Managing data and code

> The following introductory section is taken, in slightly adapted form, from Jae Yeon Kim's "Computational Thinking for Social Scientists." It is reproduced here for ease of access. To consult the full book, go to https://jaeyk.github.io/comp_thinking_social_science/.

The Command Line

What is the command line? The command line or "terminal" is what you'll recognize from Hollywood portrayals of hackers and script kids—a black screen containing single lines of code, sometimes cascading down the page.

But you all have one on your computers.

On Windows computers, you'll find this under the name "Command Prompt." See this guide on how to open.

On Mac, this is called "Terminal". See this guide. Some also prefer to use an application called "iTerm 2" but they both essentially do the same thing.

The Big Picture

As William Shotts the author of *The Linux Command Line* put it:

> graphical user interfaces make easy tasks easy, while command-line interfaces make difficult tasks possible.

Why bother using the command line?

Suppose that we want to create a plain text file that contains the word "test." If we want to do this in the command line, you need to know the following commands.

1. `echo`: "Write arguments to the standard output" This is equivalent to using a text editor (e.g., nano, vim, emacs) and writing something.

2. `> test` Save the expression in a file named test.

We can put these commands together like the following:

```
echo "sth" > test
```

Don't worry if you are worried about memorizing these and more commands. Memorization is a far less important aspect of learning programming. In general, if you don't know what a command does, just type `<command name> --help`. You can do `man <command name>` to obtain further information. Here, `man` stands for manual. If you need more user-friendly information, please consider using `tldr`.

Let's make this simple case complex by scaling up. Suppose we want to make 100 duplicates of the `test` file. Below is the one-line code that performs the task!

```
for i in {1..100}; do cp test "test_$i"; done
```

Let me break down the seemingly complex workflow.

1. `for i in {1..100}`. This is a for loop. The numbers 1..100 inside the curly braces `{}` indicates the range of integers from 1 to 100. In R, this is equivalent to for (i in 1:100) {}

2. `;` is used to use multiple commands without making line breaks. ; works in the same way in R.

3. `$var` returns the value associated with a variable. Type `name=<Your name>`. Then, type `echo $name`. You should see your name printed. Variable assignment is one of the most basic things you'll learn in any programming. In R, we do this by using ->

If you have zero experience in programming, I might have provided too many concepts too early, like variable assignment and for loop. However, you don't need to worry about them at this point. We will cover them in the next chapter.

I will give you one more example to illustrate how powerful the command line is. Suppose we want to find which file contains the character "COVID." This is equivalent to finding a needle in a haystack. It's a daunting task for humans, but not for computers. Commands are verbs. So, to express this problem in a language that computers could understand, let's first find what command we should use. Often, a simple Google or Stack Overflow search leads to an answer.

In this case, `grep` is the answer (there's also grep in R). This command finds PATTERNS in each FILE. What follows - are options (called flags): `r` (recursive), `n` (line number), `w` (match only whole words), `e` (use patterns for matching). `rnw` are for output control and `e` is for pattern selection.

So, to perform the task above, you just need one-line code: `grep -r -n -w -e "COVID''`

Quick reminders - `grep`: command - `-rnw -e`: flags - `COVID`: argument (usually file or file paths)

Let's remove (=`rm`) all the duplicate files and the original file. `*` (any number of characters) is a wildcard (if you want to identify a single number of characters, use `?`). It finds every file whose name starts with `test_`.

```
rm test_* test
```

Enough with demonstrations. What is this black magic? Can you do the same thing using a graphical interface? Which method is more efficient? I hope that my demonstrations give you enough sense of why learning the command line could be incredibly useful. In my experience, mastering the command line helps automate your research process from end to end. For instance, you don't need to write files from a website using your web browser. Instead, you can run the `wget` command in the terminal. Better yet, you don't even need to run the command for the second time. You can write a Shell script (`*.sh`) that automates downloading, moving, and sorting multiple files.

UNIX Shell

The other thing you might have noticed is that there are many overlaps between the commands and base R functions (R functions that can be used without installing additional packages). This connection is not coincident. UNIX preceded and influenced many programming languages, including R.

The following materials on UNIX and Shell are adapted from [the software carpentry](https://bids.GitHub.io/201
06-04-berkeley/shell/00-intro.html.

Unix

UNIX is an **operating system + a set of tools (utilities)**. It was developed by AT & T employees at Bell Labs (1969-1971). From Mac OS X to Linux, many of the current operation systems are some versions of UNIX. Command-line INTERFACE is a way to communicate with your OS by typing, not pointing, and clicking.

For this reason, if you're using Mac OS, then you don't need to do anything else to experience UNIX. You're already all set.

If you're using Windows, you need to install either GitBash (a good option if you only use Bash for Git and GitHub) or Windows Subsystem (highly recommended if your use case goes beyond Git and GitHub). For more information, see this installation guideline. If you're a Windows user and don't use Windows 10, I recommend installing VirtualBox.

UNIX is old, but it is still mainstream, and it will be. Moreover, the UNIX philosophy ("Do One Thing And Do It Well")—minimalist, modular software development—is highly and widely influential.

Kernel

The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs. It handles the filestore (e.g., files and directories) and communications in response to system calls.

Shell

The shell is an interactive program that provides an interface between the user and the kernel. The shell interprets commands entered by the user or supplied by a Shell script and passes them to the kernel for execution.

Human-Computer interfaces

At a high level, computers do four things:

- run programs
- store data
- communicate with each other
- interact with us (through either CLI or GUI)

The Command Line

This kind of interface is called a **command-line interface**, or CLI, to distinguish it from the **graphical user interface**, or GUI, that most people now use.

The heart of a CLI is a **read-evaluate-print loop**, or REPL: when the user types a command and then presses the enter (or return) key, the computer reads it, executes it, and prints its output. The user then types another command, and so on until the user logs off.

If you're using RStudio, you can use terminal inside RStudio (next to the "Console"). (For instance, type Alt + Shift + M)

The Shell

This description makes it sound as though the user sends commands directly to the computer and sends the output directly to the user. In fact, there is usually a program in between called a **command shell**.

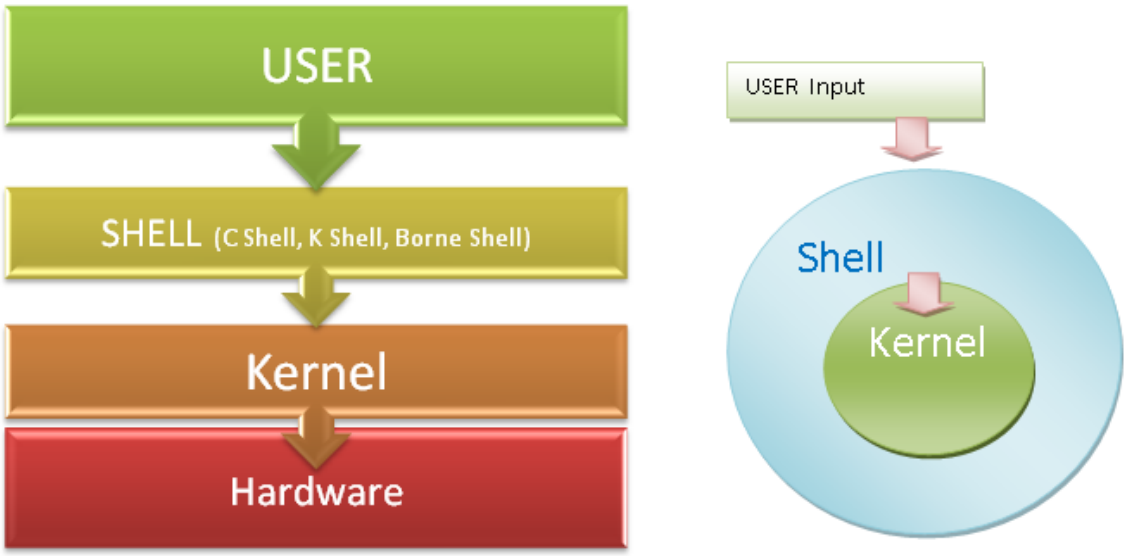

Figure 1: Source: Prashant Lakhera

What the user types go into the shell; it figures out what commands to run and orders the computer to execute them.

Note, the shell is called *the shell*: it encloses the operating system to hide some of its complexity and make it simpler to interact with.

A shell is a program like any other. What's special about it is that its job is to run other programs rather than do calculations itself. The commands are themselves programs: when they terminate, the shell gives the user another prompt ($ on our systems).

Bash

The most popular Unix shell is **Bash**, the Bourne Again Shell (so-called because it's derived from a shell written by Stephen Bourne — this is what passes for wit among programmers). Bash is the default shell on most modern implementations of **Unix** and in most packages that provide Unix-like tools for Windows.

Why Shell?

Using Bash or any other shell sometimes feels more like programming than like using a mouse. Commands are terse (often only a couple of characters long), their names are frequently cryptic, and their output is lines of text rather than something visual like a graph.

On the other hand, the shell allows us to combine existing tools in powerful ways with only a few keystrokes and set up pipelines to handle large volumes of data automatically.

In addition, the command line is often the easiest way to interact with remote machines (explains why we learn Bash before learning Git and GitHub). If you work in a team and your team manages data in a remote server, you will likely need to get access the server via something like `ssh`.

Our first command

The part of the operating system responsible for managing files and directories is called the **file system**. It organizes our data into files, which hold information, and directories (also called "folders"), which hold files or other directories.

Several commands are frequently used to create, inspect, rename, and delete files and directories. To start exploring them, let's open a shell window:

```
jae@jae-X705UDR:~$
```

Let's demystify the output above. There's nothing complicated.

- jae: a specific user name
- jae-X705UDR: your computer/server name
- `~`: current directory (`~` = home)
- `$`: a **prompt**, which shows us that the shell is waiting for input; your shell may show something more elaborate.

Type the command `whoami`, then press the Enter key (sometimes marked Return) to send the command to the shell.

The command's output is the ID of the current user, i.e., it shows us who the shell thinks we are:

```
$ whoami

# Should be your user name
jae
```

More specifically, when we type `whoami` the shell, the following sequence of events occurs behind the screen.

1. Finds a program called `whoami`,
2. Runs that program,
3. Displays that program's output, then
4. Displays a new prompt to tell us that it's ready for more commands.

Communicating to other systems

In the next unit, we'll focus on the structure of our own operating systems. But our operating systems rarely work in isolation; we often rely on the Internet to communicate with others! You can visualize this sort of communication within your own shell by asking your computer to `ping` (based on the old term for submarine sonar) an IP address provided by Google (8.8.8.8); in effect, this will test whether your Internet is working.

```
$ ping 8.8.8.8
```

Note: Windows users may have to try a slightly different alternative:

```
$ ping -t 8.8.8.8
```

(Thanks Paul Thissen for the suggestion!). Note: press ctrl + C to stop your terminal pinging!

File system organization

Next, let's find out where we are by running a `pwd` command (**print working directory**).

At any moment, our **current working directory** is our current default directory, i.e., the directory that the computer assumes we want to run commands in unless we explicitly specify something else.

Here, the computer's response is `/home/jae`, which is the **home directory**:

```
$ pwd

/home/jae
```

Additional tips

You can also download files to your computer in the terminal.