cleaning and tidying data in R

Benjamin Schmidt

2015

Cleaning Data

Overview

First, some general background.

It is a truism that for humanities data analysis (and most other types, for that matter), at least 75% of your effort will go into *cleaning* rather than analyzing data.

Most of the work of cleaning data is necessarily irregular. Different types of data all have their own their problems.

Some of the most common include the following.

Character encoding How does the text represent symbols like curly quotation marks or non-English characters like the German "eszett" (ß)? In the last decade, the world has coalesced around the **unicode standard**, which provides a coherent way to represent millions of characters from written languages around the globe. (And plenty of other places, as well; emoji, for instance, are assigned their own unicode space). ASCII, the traditional character set of 1980s American computing, is a subset of Unicode and so works well inside it. Other characters (things like curly quotes) may cause problems. If you find yourself working with a file in a different character encoding, R generally allows you to read the file in by specifying the name of the standard. You will have to Google around.

Record formats Even when you're given perfectly formatted data, you may experience some trouble reading it in.

In R, most problems will arise from conventions around **commenting** and **quoting** files. When you use read.table and encounter errors, most often you want to adjust the arguments to `comment.char=""` and `quote.char=""`, which instructs the parser to ignore the "#" sign and quotation marks. Note that some files do use quotation marks as a "quote" character, such as the New Bedford whaling data we looked at in class.

Even the simplest of characters can cause problems. If the file was created on a Macintosh, you may find that the end-of-line files aren't behaving as you would expect. This has to do with the so-called "carriage return" character (called `\r` in regular expressions, as opposed to `\n`, the standard newline.

> History moment: the reason for the peculiar behavior of the carriage return and newline has to do with the typewriter operations underlying modern computing: the `\r` symbol was supposed to move the typewriter head to the beginning of the line, while the `\n` newline character pushes the typewriter head down. In practice, `\n` is sufficient to do both; there's actually a third character, the formfeed (`\f`), that usually takes on the job of dropping the cursor down a line. You will almost never see it in use, but it can be handy to insert from time to time if you need your own record breaks.

Inconsistent category labels This is the biggest one, and one that you'll encounter in the whaling log data.

Date-time and other "type" formats What is the data type for a year? You might reasonably expect it to be a number. But in fact, you'll often find that dates.

In converting messy dates, you'll frequently have to use regular expressions.

When creating data yourself, you should use ISO 8601, which uses a "year-month-day" format.

In the example data in the excersises, you'll see an example of this in the data from the New Bedford Whaling Museum.

Data cleaning in R

Reading files: `read.table`, `read.csv`, and `scan`.

Depending on how the data you wish to read in is structured, you will typically use one of three functions in R to read it in.

`read.table` is the most common and important. It will let you read in most different sorts of data. By default, it assumes that fields are separated by a space: in the real world, you will very rarely encounter this. Instead you want generally want to read in tables separated by tabs.

How do you read in data? Generally, the information you want will be either on your hard drive (as when you format it) or on the Internet. For our first example, we'll be looking at a well-formatted CSV online.

There's some good descriptive data about people, which suggests a chance for something about bodies–measurements, physical descriptions, and ages all have interesting interplays. That will be particularly valuable if we can tie it in to some other sorts of information. Before I get into that, there are couple variables that I just want to see fuller counts on: table() in R gives the best way to do that. I'm interested in names because I could link them up to census information and because they provide some clues to ethnicity;

Reading tables and constructive failure. Often, if there is something even slightly askew about your input data, `read.table` will fail. This may be frustrating. Try to be grateful, though, instead. This failure is the first aspect of something we'll encounter many times in this class that is a general feature of data analysis: programs that don't receive *exactly* the input the expect will simply fail to work, usually "throwing" an error message of some sort.

If you have a file previously saved in the csv (comma-separated-value) format, it may be fast to read it it using the `read.csv` function. (This is simply `read.table` with a certain set of constraints.)

Workset: reading in data. You can read it on your computer by typing the following code.

```r
crews = read.csv("http://www.whalingmuseum.org/online_exhibits/crewlist/crewlist.csv")
```

So now we have data. But what's in it?

The first thing to do is simply look at. Going to the "Environment" pane in the upper right-hand corner, you can see what's in here.

Where did this data come from? Did they do a good job transcribing? Here's an example of the original source

But remember, we're doing programming here, not just browsing (plus, that list is limited to the top 1000.)

R is composed of *functions*: each of these apply on an object.

One useful function to know about is called `head`: it will show the first five elements of a data source. In R, the most common data structure is a `data.frame`; it's essentially a table where the rows correspond to observations, and the columns refer to variables. (It resembles a spreadsheet or database table).

In this data set, as you'll see, each row corresponds to an individual crew member, and the columns give information about him, such as the ship he sailed on his, his name, his rank, and so forth.

```r
head(crews)
```

The first thing to do with a new data source is run `summary`, which figures out what the different columns in your database are and gives appropriate descriptions of the types of data in each. For numbers, it gives averages; for categorical data (called 'factors') in R, it lists the most common elements.

```r
summary(crews)
```

Other formats There are many different libraries out there for reading data. R is blessed with many packages for importing them.

To read Microsoft Excel spreadsheets, it is usually easiest to simply go into Excel and click "save as CSV" to write to a more standard form.

If you don't have Excel (or LibreOffice, the free version), you can read in Excel by using the `gdata` library and its `read.xls` functions.

Cleaning Data.

There are some obvious problems with the "crews" data we need to fix for analysis.

The `$` Operator The dollar sign in R has a special meaning: it takes a part of a larger structure. With a data.frame, it takes the named column: so to see just the dates, you can type `crews$ApproximateDeparture`.

First we want to determine what kind of data the date field is. In R, the `class` function tells you the data type. With dplyr loaded, there are two ways to apply a function: using parenthesis as below, or the `%>%` construction which **chains** operations together. As you can see, both produce the same result.

```r
class(crews$ApproximateDeparture)

crews$ApproximateDeparture %>% class()
```

It is a "factor." Factors in R are categorical data; frequently, `read.table` will assume that any type of character data is a factor. (In general, it's often a good idea to supply the argument `stringsAsFactors=FALSE` to `read.table` and `read.csv`, unless you know that you need a factor for your analysis.)

We want to extract the years.

Using `gsub` to clean data.

Note that the date doesn't follow a standard form. We're going to use **regular expressions** again to clean it up. Notice the steps in the chain here. We're creating a new column called "year." Then we convert to a **character** type. Finally, we feed the output into the function `gsub`, which

Remember the function `gsub`: it is among the most important tools for data cleaning in R you will encounter. It lets you use the full power of regular expressions for find-replace operations. This regular expression is complicated–you may have to refer back to your sheet to see what's going on. Note in particular that the parenthesis are performing a **grouping** operation. `gsub` is a substitution function, so you have to tell it what to replace it with. In this case, the escaped phrase `\\1` tells R to substitute with *the first matched group*. (That's what `\\1` means: `\\2` will match the second matched group, and so forth.)

One feature of working with regular expressions in R is that you have to supply two backslash characters in front of a special character instead of one.

```
crews$year= crews$ApproximateDeparture %>% as.character %>% gsub(".*([0-9]{4}).*","\\1",.)
```

Excercises: reading and cleaning data in R

1. Is the New Bedford Crewlist data "tidy" according to Hadley Wickham's standards?
2. I've pasted below the code that fixes the date field using regular expressions. (We'll get more into just what it does over the next weeks: but an important part of coding is occasionally re-using code that you *don't* completely understand.) Add two more groups using parenthesis and more numbers: make it turn the date into the standard "1865-11-7" format.

```
crews$year= crews$ApproximateDeparture %>% as.character %>% gsub(".*([0-9]{4}).*","\\1",.)
```

3. Located in this folder, there is another file called "shiptypes.csv". It shows, the number of vessels in the database by year and ship type. It's in the folder "data": the way you represent folder names is by prefixing with a slash (or maybe a backslash on Windows.) It will probably, then be located at `data/shiptypes.csv`. Read it in, and look at it.

Fix the following code to read it in, and look at it.

```
shiptypes = read.csv("????")
```

5. That data will reveal something obviously wrong with the crewlist data. Standardize the ship names in this set using regular expressions and the `$` described above.

```
crews$rig= crews$ApproximateDeparture %>% as.character %>% gsub(".*([0-9]{4}).*","\\1",.)
```

Reformatting information: the `tidyr` package

Hadley Wickham's articles on "tidy" data lay out an extremely useful way to think about.

Exercises: tidying data

1. Is the data you read in under shiptypes `tidy`?
2. Using the `gather` and `spread` functions from the `tidyr` package, create a new data.frame that has rows corresponding to ship types and columns corresponding to years. (That is, the exact inverse of the data.frame you saw before.)
3. There's another file in the `data` directory called CESTACityData.csv. It was graciously provided by the Center for Spatial and Textual Analysis https://cesta.stanford.edu/ Read it in. We'll be working with this data set more next week.
4. This data is clearly not tidy. (Why not?) Use `gather` to turn it into tidy data, and save it as a data.frame called "tidied". (Hint: the final frame will have a column called "year". You will have to use gsub to remove Xs from the year name.)

5. Use the `?` operator to read about the function `write.csv`. Save the result to disk: we'll be exploring it more next week.

```
write.csv("???")
```

Tidy data for network visualizations.

If you are interested in network visualizations and have data like So and Long's, come to me: we'll talk about how to use tidyr to reshape some data for network analysis to be exported into a program like Gephi.