notes_gcd_week1

Derek Hall

4/30/2020

Week 1 Notes

Pipeline: Raw data -> Processing Script -> tidy data -> data analysis -> data communication

Raw Data

- Original source of the data
- Often hard to use for analysis
- Data analysis *includes* processing
- Raw data may only need to be processed once

Processed Data

- Data that is ready for analysis
- Processing can include merging, subsetting, transforming, etc.
- There may be standards for processing
- All steps should be recorded

Tidy Data

Four things you should have after tidying data:

1. The raw data
2. A tidy data set
3. A code book describing each variable and its values in the tidy data set (metadata)
4. An explicit and exact recipe you used to go from 1 to 2 to 3.

1. Raw data

The rawest form of the data you had access to. Identifiable if:

- You ran no software on the data
- You did not manipulate any of the numbers in the data
- You did not remove any data from the data set
- You did not summarize the data in any way

e.g. A binary file spit out by a machine, an unformatted Excel file from a company, a JSON dataset from API scraping, hand-entered numbers collected from microscope counting.

2. Tidy data

The cleaned data.

- Each variable you measured should be in one column.
- Each different observation should be in a different row.
- One table for each "kind" of variable.

- If there are multiple tables, they should have a column in the table that allows them to be linked together (an index/id).

tips:

- Include a row at the top of each file with variable names.
- Make variable names human readable (i.e. AgeAtDiagnosis instead of AgeDx)
- In general, save one table per file.

3. The Code Book

A file that provides important metadata about the variables contained in the raw/tidy datasets.

- Information about the variables (e.g. units) not contained in the tidy data.
- Information about the summary choices you made.
- Information about the experimental study design you used.

tips:

- A common format for this document is a word/text file.
- There should be a "Study design" that has a thorough description of how you collected the data. (e.g. How did you decide what variables to collect. What data did you exclude.)
- There must be a section called "Code book" that describes each variable and its units.

4. The instruction list

A description (i.e. recipe) for how you analyzed the raw data. Ideally a computer script. The input should be the raw data and the output should be the tidy data. There should be *no* parameters. No modifications should be necessary and it should reproduce the same results everytime.

tip: Sometimes it is not possible to script every step. In this case, write an explicit (including versions of software!) description of the steps you took to analyze the data.

Downloading files

Getting and setting your working directory.

Two commands *getwd()* and *setwd()*.

Be aware of relative versus absolute paths:

Relative: - setwd("./data") will move from current directory to the folder "data".
- setwd("../") will move up one directory level.

Absolute: - setwd("/Users/jtleek/data/") moves you to a specific directory.

Checking for and creating directories

file.exists("directoryName") will check to see if a directory exists.

dir.create("directoryName") will create a directory if it doesn't exist.

One common strategy when creating reproducible code is to check for the existence of the directories the code will need and create them if they are not present:

```
if (!file.exists("data")) {
        dir.create("data")
}
```

Getting data from the internet

download.file() Downloads a file from the internet. Helps with reproducibility even if the task can easily be done by hand. Important parameters include *url, destfile, method*.

e.g. downloading a file from the baltimore open data.

```
fileUrl <- "https://data.baltimorecity.gov/api/views/dz54-2aru/rows.csv?accessType=DOWNLOAD"

download.file(fileUrl, destfile = "./data/cameras.csv", method = "curl")
```

Files in a directory can be checked from within R using the *list.files* command.

An important component of downloading files from online is that the files might change. therefore, keeping track of when a file was downloaded is important. This can be achieved with the *date()* function:

```
dateDownloaded <- date()
dateDownloaded
```

Note:

- If the url starts with *http* you can use *download.file()*.
- If the url starts with *https* on a Windows, you may be ok. On a mac you may need to set the method to *method = "curl"*.
- Large files can take time to download. Therefore, it might be better to not hard code the file download but simply include it in the text instructions so that the download is not repeated every time the script is run.
- Be sure to record when you downloaded.

Reading local flat files

read.table() is the main function for reading data into R. It is flexible and robust, but requires more parameters. It reads the data into RAM - so big data can cause problems.

Basic Parameters: file, header, sep, row.names, nrows

Note: *read.csv* automatically sets *sep = "."* and *header = TRUE*.

Other Parameters:

- *quote*: whether there are quoted values (*quote = ""* means ignore quotes). This can be very helpful if there are quotes interspersed in the file that are confusing R (which tries to read them as character string indicators).
- *na.strings*: set the character that represents a missing value.
- *nrows*: how many rows of the file to read.
- *skip*: number of lines to skip before starting to read.

Reading Excel files

Excel files can be read using the **xlsx package** and the associated *read.xlsx()* and *read.xlsx2()* functions.

Parameters:

- *sheetIndex*: the sheet number to read.
- *header*: TRUE/FALSE define if the first row represents data headers.
- *colIndex* and *rowIndex*: define the column and row numbers to read.

Notes:

- *write.xlsx* will write out an Excel file (using the same parameters).
- *read.xlsx2* is much faster, but can be unstable when reading a subset.

- *XLConnect* has more options for reading, writing, and manipulating Excel files if you must work with Excel often.
- In general, it is advised to store your data in either a database or a comma separated file (.csv) or tab separated (.tab/.txt) as they are easier to distribute.
- May need to set mode to "wb" (write binary) when downloading from an online source.

Reading XML

XML (extensible markup language) is a format frequently used to store structured data and is particularly widely used in internet applications.

Extracting XML is the basis of most web scrapping.

Compontents:

1. *Markup*: labels that give the text structure
2. *Content*: the actual text of the document

Tags, elements, and attributes

- Tags correspond to general labels. (e.g. start tag:
  , end tag:
  , empty tag:)
- Elements are specific examples of tags (e.g. Hello, world)
- Attributes are components of the label (e.g. img src="jeff.jpg" alt ="instructor"/>)

Example XML file reading:

```
# Extract content by attributes from website
library(XML)
```

```
## Warning: package 'XML' was built under R version 3.6.3
```

```
library(RCurl)

# URL where the data resides
fileURL <- "https://www.w3schools.com/xml/simple.xml"

# We cannot use htmlTreeParse since it is https therefore use curl and getURL instead
# Instead we need to download it using getURL
xmlFile <- getURL(fileURL)

# And then parse the data into doc variable, we need to use htmlParse since this
# is an html file, but it contains embedded XML. useInternal = TRUE gets us all the
# different nodes inside that file
doc <- htmlParse(xmlFile, useInternal = TRUE)

rootNode <- xmlRoot(doc)
```

You can drill into the Xml object using subsetting.

```
rootNode[[1]][[1]][[1]]
```

```
## <food>
##   <name>Belgian Waffles</name>
##   <price>$5.95</price>
##   <description>Two of our famous Belgian Waffles with plenty of real maple syrup</description>
##   <calories>650</calories>
```

```
## </food>
```

```
rootNode[[1]][[1]][[1]][[1]]
```

```
## <name>Belgian Waffles</name>
```

You can also programmatically extract parts of the file using *xmlSapply*

```
xmlSApply(rootNode[[1]], xmlValue)
```

```
##
## "Belgian Waffles$5.95Two of our famous Belgian Waffles with plenty of real maple syrup650Strawberry
```

XPath language - */node* Top level node - *//node* Node at any level - *node[@attr-name]* Node with an attribute name - *node[@attri-name='bob']* Node with attribute name attri-name = 'bob'

more information at: http://www.stat.berkeley.edu/~statcur/Workshop2/Presentations/XML.pdf

e.g.

Get the items on the menu

```
xpathSApply(rootNode, "//name", xmlValue)
```

```
## [1] "Belgian Waffles"             "Strawberry Belgian Waffles"
## [3] "Berry-Berry Belgian Waffles" "French Toast"
## [5] "Homestyle Breakfast"
```

Get the item prices

```
xpathSApply(rootNode,"//price", xmlValue)
```

```
## [1] "$5.95" "$7.95" "$8.95" "$4.50" "$6.95"
```

More info: http://www.omegahat.net/RSXML/shortIntro.pdf

Reading JSON

JSON (Javascript Object Notation) is similar to XML in that it commonly used on the internet. Has lightweight data storage so is a common format for application programming interfaces (APIs). Structure is similar to XML but has a very different syntax.

To read and write data from and to JSONs, use the *jsonlite* package.

```
library(jsonlite)
jsonData <- fromJSON("https://api.github.com/users/jtleek/repos")
names(jsonData$owner)
```

```
##  [1] "login"               "id"                  "node_id"
##  [4] "avatar_url"          "gravatar_id"         "url"
##  [7] "html_url"            "followers_url"       "following_url"
## [10] "gists_url"           "starred_url"         "subscriptions_url"
## [13] "organizations_url"   "repos_url"           "events_url"
## [16] "received_events_url" "type"                "site_admin"
```

```
myjson <- toJSON(iris, pretty = TRUE)
cat(head(myjson[1]))
```

Further resources:

- http://www.json.org/
- http://www.r-bloggers.com/new-package-jsonlite-a-smarter-json-encoderdecoder/

Using *data.table*

Inherets from data.frame and written in C so much faster. Much faster at subsetting, group, and updating.

Typical *data.frame* method:

```
DF = data.frame(x = rnorm(9),
                y = rep(c("a", "b", "c"), each = 3),
                z = rnorm(9))
head(DF,3)
```

```
##           x y         z
## 1 -1.537169 a 0.7874885
## 2  0.199297 a 1.8448223
## 3  1.055967 a 1.6831337
```

With *data.table* method:

```
library(data.table)
```

```
## Warning: package 'data.table' was built under R version 3.6.3
```

```
DT = data.table(x = rnorm(9),
                y = rep(c("a", "b", "c"), each = 3),
                z = rnorm(9))
head(DT, 3)
```

```
##             x y          z
## 1: -2.5625593 a -1.3494053
## 2: -0.3434583 a -1.4390624
## 3: -1.3572914 a  0.5011401
```

All tables in memory can be viewed with the *tables()* command.

```
tables()
```

```
##    NAME NROW NCOL MB  COLS KEY
## 1:   DT    9    3  0 x,y,z
## Total: 0MB
```

Subsetting Rows

```
DT[2,]
```

```
##             x y         z
## 1: -0.3434583 a -1.439062
```

```
DT[c(2,3)]
```

```
##             x y          z
## 1: -0.3434583 a -1.4390624
## 2: -1.3572914 a  0.5011401
```

Subsetting Columns is modified for data.table. The argument you pass after the comma is called an expression. In R, an expression is a collection of statements enclosed in curley brackets

Expression functionality in R:

```
k = {print(10); 5}
```

```
## [1] 10
```

```
print(k)
```

```
## [1] 5
```

Because of this, in *data.table* you can pass a list of functions you want to perform.

e.g.

```
DT[, list(mean(x), sum(z))]
```

```
##            V1        V2
## 1: -0.0332783 -2.689604
```

```
DT[,table(y)]
```

```
## y
## a b c
## 3 3 3
```

Adding new columns

```
DT[,w:=z^2]
DT
```

```
##             x y          z         w
## 1: -2.5625593 a -1.3494053 1.8208946
## 2: -0.3434583 a -1.4390624 2.0709005
## 3: -1.3572914 a  0.5011401 0.2511414
## 4:  1.1150381 b -0.5956185 0.3547614
## 5:  0.4198108 b -0.7406861 0.5486159
## 6:  1.3571969 b  1.6848643 2.8387679
## 7:  0.1132091 c -1.4549264 2.1168107
## 8: -0.1646518 c  1.4300317 2.0449907
## 9:  1.1232013 c -0.7259416 0.5269912
```

CAUTION: Data table doesn't store a second copy of modified tables. So changing the original table will change subsequent iterations. Therefore, if you want to make a copy you have to explicitly make a copy with the *copy()* function.

```
DT2 <- DT
DT3 <- copy(DT)
head(DT, 3)
```

```
##             x y          z         w
## 1: -2.5625593 a -1.3494053 1.8208946
## 2: -0.3434583 a -1.4390624 2.0709005
## 3: -1.3572914 a  0.5011401 0.2511414
```

```
head(DT2, 3)
```

```
##             x y          z         w
## 1: -2.5625593 a -1.3494053 1.8208946
## 2: -0.3434583 a -1.4390624 2.0709005
## 3: -1.3572914 a  0.5011401 0.2511414
```

```
head(DT3, 3)
```

```
##             x y          z         w
## 1: -2.5625593 a -1.3494053 1.8208946
## 2: -0.3434583 a -1.4390624 2.0709005
## 3: -1.3572914 a  0.5011401 0.2511414
```

```
DT[, y := 2]

head(DT, 3)
```

```
##             x y          z         w
## 1: -2.5625593 2 -1.3494053 1.8208946
## 2: -0.3434583 2 -1.4390624 2.0709005
## 3: -1.3572914 2  0.5011401 0.2511414
```

```
head(DT2, 3)
```

```
##             x y          z         w
## 1: -2.5625593 2 -1.3494053 1.8208946
## 2: -0.3434583 2 -1.4390624 2.0709005
## 3: -1.3572914 2  0.5011401 0.2511414
```

```
head(DT3, 3)
```

```
##             x y          z         w
## 1: -2.5625593 a -1.3494053 1.8208946
## 2: -0.3434583 a -1.4390624 2.0709005
## 3: -1.3572914 a  0.5011401 0.2511414
```

Multiple operations can be performed to generate new variables.

```
DT[, m := {
        tmp <- x + z
        log2(tmp + 5)
}]

DT
```

```
##             x y          z         w         m
## 1: -2.5625593 2 -1.3494053 1.8208946 0.1217256
## 2: -0.3434583 2 -1.4390624 2.0709005 1.6859309
## 3: -1.3572914 2  0.5011401 0.2511414 2.0509713
## 4:  1.1150381 2 -0.5956185 0.3547614 2.4645166
## 5:  0.4198108 2 -0.7406861 0.5486159 2.2262387
## 6:  1.3571969 2  1.6848643 2.8387679 3.0075653
## 7:  0.1132091 2 -1.4549264 2.1168107 1.8711666
## 8: -0.1646518 2  1.4300317 2.0449907 2.6474020
## 9:  1.1232013 2 -0.7259416 0.5269912 2.4322271
```

plyr like operations can also be performed (storing logical indexes within the table)

```
DT[, a := x > 0]
DT
```

```
##             x y          z         w         m     a
## 1: -2.5625593 2 -1.3494053 1.8208946 0.1217256 FALSE
## 2: -0.3434583 2 -1.4390624 2.0709005 1.6859309 FALSE
## 3: -1.3572914 2  0.5011401 0.2511414 2.0509713 FALSE
## 4:  1.1150381 2 -0.5956185 0.3547614 2.4645166  TRUE
## 5:  0.4198108 2 -0.7406861 0.5486159 2.2262387  TRUE
## 6:  1.3571969 2  1.6848643 2.8387679 3.0075653  TRUE
## 7:  0.1132091 2 -1.4549264 2.1168107 1.8711666  TRUE
## 8: -0.1646518 2  1.4300317 2.0449907 2.6474020 FALSE
## 9:  1.1232013 2 -0.7259416 0.5269912 2.4322271  TRUE
```

```
DT[, b := mean(x+z), by = a]
DT
```

```
##             x y          z         w         m     a          b
## 1: -2.5625593 2 -1.3494053 1.8208946 0.1217256 FALSE -1.3213141
## 2: -0.3434583 2 -1.4390624 2.0709005 1.6859309 FALSE -1.3213141
## 3: -1.3572914 2  0.5011401 0.2511414 2.0509713 FALSE -1.3213141
## 4:  1.1150381 2 -0.5956185 0.3547614 2.4645166  TRUE  0.4592296
## 5:  0.4198108 2 -0.7406861 0.5486159 2.2262387  TRUE  0.4592296
## 6:  1.3571969 2  1.6848643 2.8387679 3.0075653  TRUE  0.4592296
## 7:  0.1132091 2 -1.4549264 2.1168107 1.8711666  TRUE  0.4592296
## 8: -0.1646518 2  1.4300317 2.0449907 2.6474020 FALSE -1.3213141
## 9:  1.1232013 2 -0.7259416 0.5269912 2.4322271  TRUE  0.4592296
```

groups variables by a (F vs. T), takes the mean of F and T and assigns them to the respective cases according to a.

Special Variables

.N An integer, length 1, containing the number count Allows you to easily determine how many observations for a given group are present.

e.g.

```
set.seed(123);
DT <- data.table(x = sample(letters[1:3], 1E5, TRUE))
DT[, .N, by = x]
```

```
##    x     N
## 1: c 33294
## 2: b 33305
## 3: a 33401
```

Keys

Allows you to set a default subset column.

```
DT <- data.table(x=rep(letters[1:3], each = 100), y = rnorm(300))
setkey(DT, x)
DT["a"]
```

```
##       x           y
##   1: a  0.88631257
##   2: a  2.82858132
##   3: a  2.03145429
##   4: a  1.90675413
##   5: a  0.21490826
##   6: a -0.86273413
##   7: a -2.20493863
##   8: a  0.24105923
##   9: a  1.83832419
##  10: a  0.79205468
##  11: a  0.65053469
##  12: a -1.53912061
##  13: a -0.60830053
##  14: a  0.38195644
##  15: a -1.07500044
##  16: a  0.21994264
```

```
##  17: a -0.78288781
##  18: a -1.11003346
##  19: a -1.65871456
##  20: a -0.50147343
##  21: a  1.91636375
##  22: a  1.41236645
##  23: a  0.92260986
##  24: a  1.01106201
##  25: a  0.57213026
##  26: a -0.62843126
##  27: a -0.36316140
##  28: a -1.05858811
##  29: a -0.42935803
##  30: a  0.86941467
##  31: a -0.54001647
##  32: a -1.14647747
##  33: a -0.17151840
##  34: a -0.56368340
##  35: a -0.42994346
##  36: a -1.23723779
##  37: a  0.15901329
##  38: a -1.16711067
##  39: a -0.08111944
##  40: a -0.51667953
##  41: a  0.99540703
##  42: a  0.79752142
##  43: a  0.53895224
##  44: a -1.40405605
##  45: a  0.40144065
##  46: a -0.52432237
##  47: a -0.83952146
##  48: a  0.47556591
##  49: a -0.01194696
##  50: a  0.10319780
##  51: a -0.38575415
##  52: a  1.11726438
##  53: a -0.49961390
##  54: a -0.44735091
##  55: a -0.23784512
##  56: a -0.86939374
##  57: a  1.14887678
##  58: a  0.53864996
##  59: a -0.10680992
##  60: a  0.60053649
##  61: a -1.47499445
##  62: a  0.98126964
##  63: a -0.61118738
##  64: a  0.08938648
##  65: a -0.01327227
##  66: a -0.97219341
##  67: a -0.57946225
##  68: a  0.14963144
##  69: a  0.47640689
##  70: a  0.44729682
```

```
##  71: a -0.19180956
##  72: a  0.51712710
##  73: a  0.40338273
##  74: a  1.78411385
##  75: a  0.27775645
##  76: a  0.77394978
##  77: a -2.08081928
##  78: a -0.35920889
##  79: a -0.45932217
##  80: a  0.20181947
##  81: a  0.62401138
##  82: a -0.25722981
##  83: a  0.94414021
##  84: a  0.25074808
##  85: a -0.72784257
##  86: a  0.36881323
##  87: a  0.44415068
##  88: a -1.00535422
##  89: a -0.33152471
##  90: a -0.37039325
##  91: a -0.79701529
##  92: a  0.28148559
##  93: a  0.33307250
##  94: a  0.52690325
##  95: a -0.78168949
##  96: a -0.02793948
##  97: a -1.74492339
##  98: a  0.65284209
##  99: a -0.93830821
## 100: a  0.62753159
##      x           y
```

Keys can be used to facilitate Joins

Joins

```
DT1 <- data.table(x = c("a", "a", "b", "dt1"), y = 1:4)
DT2 <- data.table(x = c("a", "b", "dt2"), z=5:7)
setkey(DT1, x)
setkey(DT2, x)
merge(DT1, DT2)
```

```
##    x y z
## 1: a 1 5
## 2: a 2 5
## 3: b 3 6
```

Reading

Use the *fread* function. Similar to *read.table* but automatically detects parameters and is much faster.