Getting and Cleaning Data

Nicola Davide D'Avanzo

10/02/2015

This document develops important informations about getting and cleaning data. In particular:

* finding and extracting raw data
* tidy data principles and how to make data tidy
* practical implementation through a range of R packages

The useful path to knowledge consists of

* raw data
* processing script
* tidy data
* data analysis
* data communication

Raw data:

* the original source of the data
* often hard to use for data analysis
* data analysis includes processing
* raw data may only need to be processed once
* raw data right format: no software on the data, not manipulated, not remove any data, not summarized.

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

4 things you should have:

* the raw data
* a tidy data set
* a code book describing each variable and its values in the tidy data set
* an explicit and exact recipe you used to go from raw data to processing script/tidy data

Tidy data:

* each variable you measure should be in one column
* each different observation of that variable should be in a differet row
* there should be one table for each kind of variable
* if you have multiple tables, they should include a column in the table that allows them to be linked
* include a row at the top of each file with variable names (header)
* make variable names human readable
* in general data should be saved in one file per table

The code book:

* information about the variables (including units) in the dataset not contained in the tidy data
* information about the summary choices you made
* information about the experimental study design you used
* a common format for this document is a Word/text file
* there should be a section called "Study design" that has a thorough description of how you collected the data
* there must be a section called "Called book" that describes each variable and its units

The instruction list:

* a computer script
* the input for the script is the raw data
* the output is the processed, tidy data
* there are no parameters to the script

Downloading files:

* get/set your working directory by *getwd()* and *setwd()*
* checking for and creating directories by *file.exists("directoryName")* and *dir.create("directoryName")*
* if the url starts with http you can use *download.file()*
* if the url starts with https on Windows you may be ok
* if the url starts with https on Mac you may need to set method ="curl"
* if the file is big, this might take a while
* be sure to record when you downloaded

Local files:

* *read.table()* is the main function for reading data into R
* flexible and robust but requires more parameters
* reads data into R (big data may be a problem)
* important parameters: *file, header, sep, row.names, nrows*.
* related to *read.csv()* and *read.csv2()*
* *quote*, you can tell R whether there are any quoted values quote="" means no quotes
* *na.strings*, set the character that represents a missing value
* *nrows*, how many rows to read in the file
* *skip*, number of lines to skip before starting to read

Excel files:

* probably the most widely used format for sharing data
* *read.xlsx()* and *read.xlsx2()* belong to xlsx package
* *colIndex* and *rowIndex*, for reading specific rows and columns
* *write.xlsx* write an Excel file
* *read.xlsx2()* is much faster but unstable for subsets
* *XLConnect* package has more functions for writing and manipulating Excel files
* It is adviced to store data in .csv or .tab/.txt as they are easier to destribute

XML files:

* particularly widely used in internet applications
* extracting XML is the basis for most web scraping
* markup, labels that give the text structure
* content, the actual text of the document
* read the file into R through the XML package
* */node* top level node
* *//node* node at any level
* *node[@attr-name]* node with an attribute name
* *node[@attr-name='bob']* node with an attribute name attr-name='bob'
* *library(XML)*
* *fileUrl<-"*[*http://*](http://)*..."*
* *doc<-xmlTreeParse(fileUrl,useInternal=TRUE)*
* *rootNode<-xmlRoot(doc)*
* *xmlName(rootNode)*
* *names(rootNode)*
* *xmlSApply(rootNode,xmlValue)*
* *xpathSApply(rootNode,"//node",xmlValue)*
* *doc<-htmlTreeParse(fileUrl,useInternal=TRUE)*
* *xpathSApply(doc,//node[@attr-name='bob'],xmlValue)*

JSON files:

* Javascript Object Notation
* common format for data from application programming interfaces (APIs)
* read the data from JSON through jsonlite package
* *library(jsonlite)*
* *jsonData<-fromJSON("*[*https://*](https://)*...")*
* *names(jsonData)* nasted objects in JSON
* *names(jsonData$...)* nasted objects in JSON
* *toJSON(...,pretty=TRUE)* writing data frames to JSON
* *fromJSON(...)* convert back to JSON

Using *data.table*:

* all functions that accept data.frame work on data.table
* much faster at subsetting, group and updating as written in C
* create data tables just like data frames
* *library(data.table)*
* *DT=data.table(x=...,y=,...,z=...)*
* *tables()* in order to see all data tables in memory
* *DT[DT$y=="a",]*
* *DT[,list(mean(x),sum(z))]*
* *:=* to assign a column specified values
* *DT[,.N,by=x]*
* *.N* used to count
* *by=...* perform a calculation just for the specified attributes
* *setkey(DT,x)* to sort a data.table and marks it as sorted
* *setkey()* also useful to join data tables
* *fread()* similar to *read.table* but faster and more convenient (all controls such as *sep, colClasses, nrows* are automatically detected)

*Reading MySQL*:

* widely used in internet based applications
* data are structured in: databases, tables within databases, fields within tables
* each row is called a record
* package RMySQL
* connecting databases *ucscDb<-dbConnect(MySQL(),user="",host="")*
* disconnecting *dbDisconnect()*
* listing databases *result<-dbGetQuery(ucscDb,"")*
* listing tables *allTables<-dbListTables()*
* get dimension of a specific table *allFields<-dbListFields()*
* read from the table *dbReadTable()*
* select a specific subset *query<-dbSendQuery(); fetch(query); quantile()*

*Reading HDF5*

* used for storing large data sets
* hierarchical data format
* *groups* containing zero or more data sets and metadata (*group header* and *group symbol table*)
* datasets multidimensional array of data elements with metadata (*header* and *data array*)
* R HDF5 package *library(rhdf5)*
* create file *created = h5createFile("file.h5")*
* create groups *created = h5createGroup("file.h5","group")*
* list file groups *h5ls("file.h5")*
* write to groups *h5write(objet,"file.h5","group")*
* assign metadata to object in a group *attr(object, "metadata")*
* write a dataset *df = data.frame(); h5write(df,"file.h5","df")*
* reading data *readGroup = h5read("file.h5","group")*
* writing and reading chunks *h5write(c(12,13,14),"file.h5","group",index=list(1:3,1)); h5read("file.h5","group")*

*Reading Data from the Web*

* webscraping: programmatically extracting data from the HTML code of websites.
* getting data off webpages: *con = url("*[*http://*](http://)*..."); htmlCode = ReadLines(con); close(con); htmlCode*
* Parsing with XML: *library(XML); url<-"*[*http://*](http://)*..."; html<-htmlTreeParse(url,useInternalNodes=TRUE); xpathSApply(html,"//...",xmlValue)*
* GET from the httr package: *library(httr); html2=GET(url); content2=content(html2,as="text"); parseHtml=htmlParse(content2,asText=TRUE);xpathSApply(parseHtml,"//...",xmlValue)*
* accessing website with passwords: *pg1=GET("*[*http://*](http://)*...",authenticate("user","passwd"))*
* using handles *google=handle("*[*http://google.com*](http://google.com)*"); pg1=GET(handle=google,path='/'); pg2=GET(handle=google,path="search")*

*Reading data from API's*

* Accessing Twitter from R: *myapp=oauth\_app("twitter",key="yourConsumerKeyHere",secret="yourConsumerSecretHere"); sig=sign\_oauth1.0(myapp,token="yourTokenHere",token\_secret="yourTokenSecretHere");homeTL=GET("*[*https://*](https://)*...")*
* Converting the json object: *json1=content(homeTL); json2=jsonlite::fromJSON(toJSON(json1))*
* httr allows GET, POST, PUT, DELETE requests if you are authorized
* you can authenticate with a user name or a password
* most modern APIs use something like oauth
* httr works well with Facebook, Google, Twitter, Github, etc...

*Interacting more directly with files*

* file - open a connection to a text file
* url - open a connection to a url
* gzfile - open a connection to a .gz file
* bzfile - open a connection to a .bz2 file
* ?connections for more information
* remember to close connection

*Foreign package*

* Loads data from Minitab, S, SAS, SPSS, Stata, Systat
* basic functions *read.foo*
* *read.arff* (Weka)
* *read.dta* (Stata)
* *read.mtp* (Minitab)
* *read.octave* (Octave)
* *read.spss* (SPSS)
* *read.xport* (SAS)

*Reading images*

* jpeg
* readbitmap
* png
* EBImage (Bioconductor)

*Reading GIS data*

* rdgal
* rgeos
* raster

*Reading music data*

* tuneR
* seewave

*Subsetting and sorting*

* logicals ands and ors: *X[(X$ var1 & X$ var2),]*, *X[(X$ var1 | X$ var2),]*
* *X[which(X$var2>8),]*
* *sort(...,decreasing=TRUE,na.last=TRUE)*
* *X[order(X$ var1,X$ var3),]*
* ordering with plyr: *library(plyr)*
* *arrange (X,desc(var1))*

*Summarizing Data*

* Look at a bit of the data: *head()*, *tail()*
* Make summary: *summary()*
* Mpre in depth information: *str()*
* Quantile of quantitative variables: *quantile(X,na.rm=TRUE,probs=c(0.2,0.5,0.8))*
* make table: *table(X,useNA="ifany")*, *table(X,Y)*
* check for missing values: *sum(is.na(X))*, *any(is.na(X))*, *all(X)*
* row and column sums: *colSums(is.na()X)*
* values with specific characteristics: *table(X %in% c("156","45424"))*, *X[X %in% c("156","45424")]*
* cross tabs: *xt<-xtabs(x~y+z,data=DF)*
* flat tables: *xt=xtabs(x~.,data=DF)*, *ftable(xt)*
* size of a dataset: *object.size(X)*, *print(object.size(X),units="Mb")*

*Creating new variables*

* often the raw data won't have the column you are looking for
* you will need to transform the data to get the values you would like
* usually you will add those values to the data frames you are working with
* common variables to create: missingness indicators, "cutting up" quantitative variables, applying transforms.
* creating sequences: *seq()*
* subsetting variables: *X$a=X %in% c("x","y")*
* creating binary variables *X$a=ifelse(X<0,TRUE,FALSE)*
* creating categorical variables: *Xa,breaks=quantile(X$a))*
* easier cutting: *library(Hmisc); Xa,g=4)*
* creating factor variables: *Xa)*
* levels of factor variables: *yesno<-sample(c("yes","no"),size=10,replace=TRUE); yesnofac=factor(yesno,levels=c("yes","no"));relevel(yesnofac,ref="yes")*

*Reshaping data*

* The goal is tidy data: each variable forms a column, each observation forms a row, each table/file stores data about one kind of observation.
* *library(reshape2)*
* melting data frames: *melt()*
* casting data frames: *dcast(X,a~b,mean)*
* averaging values: *tapply(Xb,sum)*
* split: *split(Xb)*
* apply: *lapply(X,sum)*
* combine: *unlist(X); sapply(X,sum)*
* plyr package: \*ddply(X,.(a),summarize,sum=sum(count))
* creating a new variable: *ddply(X,.(a),summarize,sum=ave(count,FUN=sum))*

*Managing data frames with dplyr*

* dplyr is an optimized and distilled version of *plyr* package
* dplyr greatly simplifies existing functionality in R
* dplyr is very fast, as many key operations are coded in C++
* *select()*: return a subset of the columns of a data frame
* *filter()*: extract a subset of rows from a data frame based on logical conditions
* *arrange()*: reorder rows of a data frame
* *rename()*: rename variables in a data frame
* *mutate()*: add new varibles/columns or transform existing variables
* *summarise/summarize()*: generate summary statistics of different variables in the data frame, possibly within strata

*Marging data*

* *merge(X,Y,by.x="a",by.y="b",all=TRUE)*
* merge all common column names: *intersect(names(X),names(Y))*
* using *join()* in the playr package: *arrange(join(df1,df2),id)*
* if you have multiple data frames: *dfList=list(df1,df2,df3);join\_all(dfList)*

*Editing text variables*

* *tolower()*: translate characters in character vectors, in particular from upper to lower case or vice versa
* *strsplt()*: split the elements of a character vector x into substrings according to the matches to substring split within them