Getting and Cleaning Data: Week 2

Video 2-1: Reading from MySQL

mySQL

- Free and widely used open source database software
- Widely used in internet-based apps
- Data are structured in
    - Databases
    - Tables within databases
    - Fields within tables
- Each row is called a record.

http://en.wikipedia.org/wiki/MySQL http://www.mysql.com/

Example structure

Graphic showing six database tables, with their fields, and relationships between the tables (link goes to graphic)

http://dev.mysql.com/doc/employee/en/sakila-structure.html

Step 1: Install MySQL

http://dev.mysql.com/doc/refman/5.7/en/installing.html

Step 2: Install RMySQL

- On a Mac: install.packages("RMySQL")
- On Windows:
    - Official instructions: http://biostat.mc.vanderbilt.edu/wiki//Main/RMySQL (may be useful for Mac users as well)
    - Potentially useful guide: http://www.ahschulz.de/2013/07/23/installing-rmysql-under-windows

My note: Also look in the forums, many issues have arisen, many people have tried to help

Example: UCSC database

http://genome.ucsc.edu/

UCSC MySQL

This link has info on how to connect to the database:

http://genome.ucsc.edu/goldenPath/help/mysql.html

Connecting and listing databases

(The following is not an R code block and so won't execute.)

- Important to disconnect from db as soon as you're done
- The value returned should be TRUE, which indicates that the query executed.

```
ucscDb <- dbConnect(MySQL(), user="genome", host="genome-mysql.cse.ucsc.edu")
result <- dbGetQuery(ucscDb, "show databases;"); dbDisconnect(ucscDb);

[1] TRUE

result

              Database
1   information_schema
2              ailMel1
3              allMis1
...
```

Connecting to hg19 and listing tables

The following gets a list of tables in a specific database in the server.

```
hg19 <- dbConnect(MySQL(),user="genome",db="hg19",host="genome-mysql.cse.ucsc.edu")
allTables <- dbListTables(hg19)
length(allTables)

[1] 10949

allTables[1:5]

[1] "HInv"    "HInvGeneMrna"    "acembly"   "acemblyClass"   "acemblyPep"
```

Get dimensions of a specific table

- dbListFields returns a list of the field names in a table
- dbGetQuery runs a query

```
dbListFields(hg19,"affyU133Plus2")

[1]  "bin"         "matches"    "misMatches"  "repMatches"   "nCount"   "qNumInsert"
[7]  "qBaseInsert" "tNumInsert" "tBaseInsert" "strand"       "qName"    "qSize"
[13] "qStart"      "qEnd"       "tName"       "tSize"        "tStart"   "tEnd"
[19] "blockCount"  "blockSizes"  "qStarts"     "tStarts"

dbGetQuery(hg19, "select count(*) from affyU133Plus2")

   count(*)
1     58463
```

Read from the table

dbReadTable reads all of the data from a table into a data frame. Caution is advised here, the table may have a huge number of rows.

```
affyData <- dbReadTable(hg19, "affyU133Plus2")
head(affyData)
# too lazy to type it all, but this shows the first several rows of all columns in the table
```

Select a specific subset

The difference between dbGetQuery and dbSendQuery is that the former submits the query, fetches all output records, and clears the result set all in one step, but the latter does not extract any records; you need to use fetch to do that.

The first fetch statement below gets all records. The second requests only the first 10 rows (n=10).

The code below appears to be invalid. It shows two fetch commands run on a single query, and I don't think you can do that.

```
query <- dbSendQuery(hg19, "select * from affyU133Plus2 where misMatches between 1 and 3")
affyMis <- fetch(query); quantile(affyMis$misMatches)

  0%  25%  50%  75% 100%
   1    1    2    2    3

affyMisSmall <- fetch(query,n=10); dbClearResult(query);

[1] TRUE

dim(affyMisSmall)

[1] 10 22
```

Don't forget to close the connection!

```
dbDisconnect(hg19)

[1] TRUE
```

Further resources

- RMySQL vignette: http://cran.r-project.org/web/packages/RMySQL/RMySQL.pdf
- List of commandes http://www.pantz.org/software/mysql/mysqlcommands.html
    - Do not, do not, delete, add or join things from ensemble. Only select.
    - In general be careful with mysql commands
- A nice blog post summarizing some other commands: http://www.r-bloggers.com/mysql-and-r

Below is a simple example from my local server. Note that for each fetch command, I first need a separate dbSendQuery.

```
## Loading required package: DBI
```

```
## MYSQL_HOME defined as C:\Program Files\MySQL\MySQL Server 5.6
```

```
##   Code        Name     Continent                 Region
## 1  ABW       Aruba North America              Caribbean
## 2  AFG Afghanistan          Asia Southern and Central Asia
## 3  AGO      Angola        Africa            Central Africa
## 4  AIA    Anguilla North America              Caribbean
## 5  ALB     Albania        Europe          Southern Europe
## 6  AND     Andorra        Europe          Southern Europe
```

```
## [1] TRUE
```

```
##    Code                 Name     Continent                 Region
## 1   ABW                Aruba North America              Caribbean
## 2   AFG          Afghanistan          Asia Southern and Central Asia
## 3   AGO               Angola        Africa            Central Africa
## 4   AIA             Anguilla North America              Caribbean
## 5   ALB              Albania        Europe          Southern Europe
## 6   AND              Andorra        Europe          Southern Europe
## 7   ANT Netherlands Antilles North America              Caribbean
## 8   ARE United Arab Emirates          Asia             Middle East
## 9   ARG            Argentina South America           South America
## 10  ARM              Armenia          Asia             Middle East
```

```
## [1] TRUE
```

```
## [1] TRUE
```

```
library(RMySQL)
world <- dbConnect(MySQL(),user="root",password="yourPassword",db="world",host="localhost")
query <- dbSendQuery(world, "select * from country")
country <- fetch(query);
head(country[,1:4])
dbClearResult(query)
query <- dbSendQuery(world, "select * from country")
country2 <- fetch(query,n=10)
country2[,1:4]
dbDisconnect(world)
```

Video 2-2: Reading from HDF5

HDF5

- Used for storing large data sets
- Supports storing a range of data types
- Hierarchical data format (HDF)
- *groups* containing zero or more data sets and metadata
    - Have a *group header* with group name and list of attributes
    - Have a *group symbole table* with a list of objects in group
- *datasets* multidimensional array of data elements with metadata
    - Have a *header* with name, datatype, dataspace and storage layout
    - Have a *data array* with the data

http://www.hdfgroup.org

R HDF5 package

Unusual method needed to install:

```
source("http://bioconductor.org/biocLite.R")
biocLite("rhdf5")
```

- This will install packages from Bioconductor (http://bioconductor.org) primarily used for genomics but also has good “big data” packages
- Can be used to interface with hdf5 data sets
- This lecture is modeled very closely on the rhdf5 tutorial that can be found here: http://www.bioconductor.org/packages/release/bioc/vignettes/rhdf5/inst/doc/rhdf5.pdf

I am on Windows 7, and a prompt appeared asking me to approve updating several other packages. I entered “a” (for “all”) and the process failed, complaining that it could not write to the corresponding directories. I restarted R as an administrator and tried again, and this time it worked.

NOTE: The code below creates a file example.h5 in the same directory as this Rmd file, but it will first check for the existence of a file with that name and **delete** it if found.

```
library(rhdf5)
# my addition: delete the file, if it already exists
if (file.exists("example.h5")) {
    file.remove("example.h5")
}
```

```
## [1] TRUE
```

```
created = h5createFile("example.h5")
created
```

```
## [1] TRUE
```

Create groups

- The h5createGroup funciton manipulates an existing HDF5 file, creating groups inside of it.
- The h5ls command is a bit like the bash ls command, except that it lists the groups (which look like paths) inside an HDF5 file.

Note that if the groups already exist in the HDF5 file, then they are not overwritten; instead the output indicates that they already exist.

```
created = h5createGroup("example.h5", "foo")
created = h5createGroup("example.h5", "baa")
created = h5createGroup("example.h5", "foo/foobaa")
h5ls("example.h5")
```

```
##   group   name     otype dclass dim
## 0     /    baa H5I_GROUP
## 1     /    foo H5I_GROUP
## 2  /foo foobaa H5I_GROUP
```

Write to groups

- The h5write command writes any R data structure to a specified file and group

```
A <- matrix(1:10, nr = 5, nc = 2)
h5write(A, "example.h5", "foo/A")
B = array(seq(0.1, 2, by = 0.1), dim = c(5, 2, 2))
attr(B, "scale") <- "liter"
h5write(B, "example.h5", "foo/foobaa/B")
h5ls("example.h5")
```

```
##          group   name       otype  dclass       dim
## 0            /    baa   H5I_GROUP                  
## 1            /    foo   H5I_GROUP                  
## 2         /foo      A H5I_DATASET INTEGER     5 x 2
## 3         /foo foobaa   H5I_GROUP                  
## 4 /foo/foobaa      B H5I_DATASET   FLOAT 5 x 2 x 2
```

Write a data set

The following example creates a data frame and writes it to the top of the HDF5 file's hierarchy, using the name “df”.

```
df = data.frame(1L:5L, seq(0, 1, length.out = 5), c("ab", "cde", "fghi", "a",
    "s"), stringsAsFactors = F)
h5write(df, "example.h5", "df")
h5ls("example.h5")
```

```
##          group   name       otype   dclass       dim
## 0            /    baa   H5I_GROUP                   
## 1            /     df H5I_DATASET COMPOUND         5
## 2            /    foo   H5I_GROUP                   
## 3         /foo      A H5I_DATASET  INTEGER     5 x 2
## 4         /foo foobaa   H5I_GROUP                   
## 5 /foo/foobaa      B H5I_DATASET    FLOAT 5 x 2 x 2
```

Reading data

The h5read function reads from a specified point in an HDF5 file. I've added to the video example below to show that you can read not only the data structures you wrote into the file, but also the paths. So “foo” is a list, containing A (a matrix) and foobaa (a subpath in the HDF5 file, which in R will another list).

```
readA = h5read("example.h5", "foo/A")
readB = h5read("example.h5", "foo/foobaa/B")
readdf = h5read("example.h5", "df")
readA
```

```
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

```
foo = h5read("example.h5", "foo")
class(foo)
```

```
## [1] "list"
```

```
names(foo)
```

```
## [1] "A"      "foobaa"
```

```
class(foo$foobaa)
```

```
## [1] "list"
```

Writing and reading chunks

We can write to or read from subsets of data structures in HDF5 files.

```
h5write(c(12, 13, 14), "example.h5", "foo/A", index = list(1:3, 1))
h5read("example.h5", "foo/A")
```

```
##      [,1] [,2]
## [1,]   12    6
## [2,]   13    7
## [3,]   14    8
## [4,]    4    9
## [5,]    5   10
```

Notes and further resources

- hdf5 can be used to optimizing reading/writing from disk in R
- The rhdf5 tutorial: http://www.bioconductor.org/packages/release/bioc/vignettes/rhdf5/inst/doc/rhdf5.pdf
- The HDF group has information on HDF5 in general: http://www.hdfgroup.org/HDF5/

Video 2-3: Reading data from the web

Webscraping

Webscraping: Programmatically extracting data from the HTML code of websites

- It can be a great way to get data
- Many websites have information you may want to read programmatically
- In some cases this is against terms of service of site
- Attempting to read too many pages too quickly can get your IP address blocked

http://en.wikipedia.org/wiki/Web_scraping

Example: Google scholar page

http://scholar.google.com/citations?user=HI-I6C0AAAAJ&hl=en

Getting data off webpages: readLines()

The readLines function puts all of the contents of a web page into a single character vector.

```
con = url("http://scholar.google.com/citations?user=HI-I6C0AAAAJ&hl=en")
htmlCode = readLines(con)
```

```
## Warning: incomplete final line found on
## 'http://scholar.google.com/citations?user=HI-I6C0AAAAJ&hl=en'
```

```
close(con)
substr(htmlCode, start = 1, stop = 1000)
```

```
## [1] "<!DOCTYPE html><html><head><title>Jeff Leek - Google Scholar Citations</title><meta
name=\"robots\" content=\"noarchive\"><meta http-equiv=\"Content-Type\" content=\"text/html;charset=ISO-
8859-1\"><meta http-equiv=\"X-UA-Compatible\" content=\"IE=Edge\"><meta name=\"format-detection\"
content=\"telephone=no\"><link rel=\"canonical\" href=\"http://scholar.google.com/citations?user=HI-
I6C0AAAAJ&amp;hl=en\"><style type=\"text/css\" media=\"screen,
projection\">html,body,form,table,div,h1,h2,h3,h4,h5,h6,img,ol,ul,li,button{margin:0;padding:0;border:0;}table{bor
collapse:collapse;border-width:0;empty-cells:show;}#gs_top{position:relative;min-
width:964px;_width:expression(document.documentElement.clientWidth<966?\"964px\":\"auto\");-webkit-tap-
highlight-color:rgba(0,0,0,0);}#gs_top>*:not(#x){-webkit-tap-highlight-
color:rgba(204,204,204,.5);}.gs_el_ph #gs_top,.gs_el_ta #gs_top{min-
width:300px;_width:expression(document.documentElement.clientWidth<302?\"300px\":\"auto\");}body,td{font-
size:13px;font-family:Arial,sans-seri"
```

```
class(htmlCode)  # character vector
```

```
## [1] "character"
```

```
length(htmlCode)  # length is 1, there is only one element in the vector
```

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

```
nchar(htmlCode)  # number of characters in the vector
```

```
## [1] 74509
```

Parsing with XML

More convenient is using the XML library's htmlTreeParse function, against which we can use xpathSApply. As before, it helps to know XPath.

```
library(XML)
url <- "http://scholar.google.com/citations?user=HI-I6C0AAAAJ&hl=en"
html <- htmlTreeParse(url, useInternalNodes = T)
xpathSApply(html, "//title", xmlValue)
```

```
## [1] "Jeff Leek - Google Scholar Citations"
```

```
xpathSApply(html, "//td[@id='col-citedby']", xmlValue)
```

```
##  [1] "Cited by" "404"      "290"      "267"      "178"      "152"
##  [7] "136"      "134"      "126"      "117"      "45"       "33"
## [13] "32"       "30"       "23"       "15"       "15"       "12"
## [19] "12"       "9"        "7"
```

GET from the httr package

This is an alternative to using htmlTreeParse. It is preferable when a website requires authorization, as shown in next section.

- The GET function executes an HTTP GET request against a URL. It returns a “response” object which will contain response headers, response code, etc., as well as the HTML.
- The content function extracts just the HTML from a response. The as=“text” means that the value returned is a character vector.
- The htmlParse function (note: from the XML library) parses a character vector and returns the HTML.

```
library(httr)
html2 = GET(url)
class(html2)
```

```
## [1] "response"
```

```
content2 = content(html2, as = "text")
class(content2)
```

```
## [1] "character"
```

```
parsedHtml = htmlParse(content2, asText = TRUE)
xpathSApply(parsedHtml, "//title", xmlValue)
```

```
## [1] "Jeff Leek - Google Scholar Citations"
```

Accessing websites with passwords

The following requires authentication, so returns a 401. My code below also demonstrates that the response is a list, and one of its members is status_code, which contains the 401. So you can programmatically check for a 200 code, which means everything is OK.

```
pg1 = GET("http://httpbin.org/basic-auth/user/passwd")
pg1
```

```
## Response [http://httpbin.org/basic-auth/user/passwd]
##   Status: 401
##   Content-type:
##
```

```
names(pg1)
```

```
## [1] "url"         "handle"      "status_code" "headers"     "cookies"
## [6] "content"     "times"       "config"
```

```
pg1$status_code
```

```
## [1] 401
```

http://cran.r-project.org/web/packages/httr/httr.pdf

Now authenticate:

```
pg2 = GET("http://httpbin.org/basic-auth/user/passwd", authenticate("user",
    "passwd"))
pg2$status_code
```

```
## [1] 200
```

```
pg2
```

```
## Response [http://httpbin.org/basic-auth/user/passwd]
##   Status: 200
##   Content-type: application/json
## {
##   "user": "user",
##   "authenticated": true
## }
```

```
names(pg2)
```

```
## [1] "url"         "handle"      "status_code" "headers"     "cookies"
## [6] "content"     "times"       "config"
```

Using handles

A handle is a way of caching a URL and login, along with information such as cookies. So it represents a web session. This is important if you need to access the URL again without re-authenticating, or having the cookies overwritten. A handle will apply to everything on a site that is defined as being within a single web session. That generally (but not always) means all subpaths below the path you pass to the handle, and generally (but not always) excludes anything outside of that path, and its children.

Note that, as far as I can tell, authentication does not happen with the handle function. You first define a handle, and then use GET to authenticate, using the handle. But because the handle can retain cookies, subsequent use of that handle will not require re-authentication.

```
google = handle("http://google.com")
pg1 = GET(handle = google, path = "/")
pg2 = GET(handle = google, path = "search")
```

Notes and further resources

- R Bloggers has a number of examples of web scraping: http://www.r-bloggers.com/?s=Web+Scraping
- The httr help file has useful examples: http://cran.r-project.org/web/packages/httr/httr.pdf
- See later lectures on APIs

Video 2-4: Reading from APIs

Application programming interfaces

Many sites have public APIs that you can access

https://dev.twitter.com/docs/api/1/get/blocks/blocking

Creating an application

First, create an account. This usually means a developer account, beyond or in addition to a user account. Then you create an application.

My notes on creating an application:

- The Name must be unique across all Twitter users. So if you try “Test”, that won't work.
- The description must be at least 10 characters.
- After your app is created, click the “API Keys” tab", and click on the “Create token access” button. It may take a little while, but eventually this page will show “Access token” and “Access token secret” fields.

After account is created, the settings are a bit different from the video (at least for me).

- The four values you want are under the “API Keys” tab of your application.
- The video's “consumer key” is “Access token”
- The video's “consumer secret” is “Access token secret”

```
myapp = oauth_app("twitter", key="yourAccessToken",secret="yourAccessTokenSecret")
sig = sign_oauth1.0(myapp, token="yourAPIKey", token_secret="yourAPISecret")
```

The following loads the content of homeTL into json1, but the result is created using the fromJSON function from the “R JSON IO” package (that's what video says), which returns a structured R object that is not very easy to read. So the jsonlite's fromJSON and toJSON functions are used to create a more user-friendly representation of the data (it is put into a data frame).

Note that there is an error in the video. It uses jsonlite::fromJSON, in order to avoid having to load the jsonlite library first, but then it references the toJSON function with prepending it with jsonlite:: which indicates that when the author made the video, he must have already had the json lite library loaded. So I've simply loaded it in the following snippet.

```
library(jsonlite)
homeTL = GET("https://api.twitter.com/1.1/statuses/home_timeline.json", sig)
json1 = content(homeTL)
json2 = fromJSON(toJSON(json1))
json2[1, 1:4]
```

```
##                       created_at        id              id_str
## 1 Sun May 11 14:41:25 +0000 2014 4.655e+17 465501937341706240
##
text
## 1 Separatist referendum in #Ukraine. Vote yes, you get roses. Vote no, Molotov cocktails.
http://t.co/LZD1BZOIak http://t.co/33ycqSsdR7
```

How did I know what URL to use?

https://dev.twitter.com/docs/api/1.1/get/search/tweets

Look at the “Resource URL” line on this page.

NOTE: When I do this, the URL I see is:

https://api.twitter.com/1.1/search/tweets.json

But note: The screenshot in the video has a URL of:

https://dev.twitter.com/docs/api/1.1/get/statuses/home_timeline

There are actually several different URLs you can use, listed at:

https://dev.twitter.com/docs/api/1.1

Also note that, whichever URL you choose, the docs page will list other parameters you can send with your request, for example, to determine how many tweets to return.

In general look at the documentation

https://dev.twitter.com/docs/api/1.1/overview

- 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 (tokens)
- httr works well with Facebook, Google, Twitter, Github etc.

Video 2-5: Reading from other sources

There is a package for that

- Roger has a nice video on how there are R packages for most things that you will want to access

- Here I'm going to briefly review a few useful packages
- In general the best way to find out if the R package exists is to Google “data storage mechanims R package”, e.g., “MySQL R package”

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
- read.fwf hint-hint-wink-wink: Reads from a fixed-width file
- ?connections for more information
- **Remember to close connections**

foreign package

- Loads data from Minitab, S, SAS, SPSS, Stata, Systat
- Basic functions:
  - read.arff (Weka)
  - read.dta (Stata)
  - read.mtp (Minitab)
  - read.octave (Octave)
  - read.spss (SPSS)
  - read.xport (SAS)
- See the help page for more details: http://cran.r-project.org/web/packages/foreign/foreign.pdf

Examples of other database packages

- RPostgreSQL provides DBI-compliant db connection from R
  - Tutorial: https://code.google.com/p/rpostgresql/
  - Help: http://cran.r-project.org/web/packages/RPostgreSQL/RPostgresSQL.pdf
- RODBC provides interfaces to multiple databases including PostgresQL, MYSQL, MS-Access and SQLite
  - Tutorial: http://cran.r-project.org/web/packages/RODBC/vignettes/RODB C.pdf
  - Help: http://cran.r-project.org/web/packages/RODBC/RODBC.pdf
- RMongo: http://cran.r-project.org/web/packages/RMong/RMongo.pdf
  - Also: http://www.r-bloggers.com/r-and-mongodb

Reading images

- jpeg: http://cran.r-project.org/web/packages/jpeg/index.html
- readbitmap: http://cran.r-project.org/web/packages/readbitmap/index.html
- png: http://cran.r-project.org/web/packages/png/index.html
- EBImage (Bioconductor): http://www.bioconductor.org/packages/2.13/biox/html/EBImage.html

Reading GIS data

- rdgal: http://cran.r-project.org/web/packages/rdgal/index.html
- rgeos: http://cran.r-project.org/web/packages/rgeos/index.html
- raster: http://cran.r-project.org/web/packages/raster/index.html

Reading music (MP3) data

- tuneR: http://cran.r-project.org/web/packages/tuneR/
- seewave: http://rug.mnhn.fr/seewave/