Lecture 12: Transforming and Reshaping Data

36-350

6 October 2014

In Previous Episodes

- Accessing vectors, arrays, and data frames
- Applying a function across a vector, array, or data frame
- Extracting data values from more-or-less formatted text
- Functions to automate repetitive tasks

Agenda

- Review of selective access
- Review of applying functions
- Lossless vs. lossy transformations
- Common transformations of numerical data
- Re-ordering data frames
- Merging data frames

Not Our Only Example, Really

```
data("cats",package="MASS")
```

Access Tricks

Problem: get the positions in a vector / columns in a matrix / rows in a dataframe matching some condition

- Vector of Boolean indicators

```
head(cats$Hwt[cats$Sex=="M"])
```

```
## [1]  6.5  6.5 10.1  7.2  7.6  7.9
```

```
head(cats[cats$Sex=="M","Hwt"])
```

```
## [1]  6.5  6.5 10.1  7.2  7.6  7.9
```

- N.B., `cats$Sex=="M"` is a Boolean vector, as long as `cats$Sex`

Access Tricks (cont'd.)

- Vector of index numbers

```
training_rows <- sample(1:nrow(cats),size=nrow(cats)/2)
head(training_rows)
```

```
## [1]  79  62 133  70  16  82
```

```
cats.trained <- cats[training_rows,]
head(cats.trained)
```

```
##     Sex Bwt  Hwt
## 79    M 2.7  8.0
## 62    M 2.4  7.9
## 133   M 3.5 15.6
## 70    M 2.5 11.0
## 16    F 2.2  9.7
## 82    M 2.7  9.6
```

Access Tricks (cont'd.)

- Vectors of Booleans and vectors of indices can be stored and re-used

```
males <- cats$Sex=="M"
```

- See also `apply` tricks below

Access Tricks: Don't Do These

- Non-binary, non-integer vectors do not make good indices; don't say

```
movies$gross[movies$genre]
```

if you are trying to get all the gross revenues of some movies of a certain genre

- Loops are a last resort, not a first; don't say

```
for (i in 1:nrow(movies)) {
  if (movies$genre[i]=="comedy") {
  gross.comedy <- c(gross.comedy, movies$gross[i])
  }
}
```

Access Tricks: Don't Do These (cont'd.)

- In either case, say

```
movies$gross[movies$genre=="comedy"]
```

or

```
movies[movies$genre=="comedy","gross"]
```

Apply Tricks

- Lots of functions will *automatically* apply themselves to each element in a vector or dataframe; they are **vectorized**

```
dim(is.na(cats)) # checks each element for being NA
```

```
## [1] 144   3
```

- Math functions are vectorized

```
mean(log(cats$Hwt))
```

```
## [1] 2.339
```

Apply Tricks (cont'd.)

- Distribution-related functions are vectorized

```
rnorm(n=5,mean=-2:2,sd=(1:5)/5)
```

```
## [1] -1.81837 -0.82949  0.04215  0.45532  1.35941
```

- Lots of the text functions vectorize across target strings, not patterns (e.g., `grep`, `regexp`)
- If the function doesn't, or that's not quite what you want, turn to the `apply` family of functions

Apply Tricks: Vectors

- Apply the same function to every element in a vector: `sapply` or `lapply`

```
mean.omitting.one <- function(i,x) {mean(x[-i])}
jackknifed.means <- sapply(1:nrow(cats),mean.omitting.one,x=cats$Bwt)
length(jackknifed.means)
```

```
## [1] 144
```

```
sd(jackknifed.means)
```

```
## [1] 0.003394
```

- `sapply` tries to return a vector or an array (with one column per entry in the original vector)
- If that doesn't make sense, use `lapply`, which just returns a list

Apply Tricks: Rows

- Apply the same function `FUN` to every row of an array or dataframe `X`: `apply(X,1,FUN)`

```
rows_with_NAs <- apply(is.na(movies),1,any)
```

- `apply` tries to return a vector or an array; will return a list if it can't
- `apply` assumes `FUN` will work on a row of `X`; might need to write a little adapter function to make that true

```
# Make 3rd and 5th cols. of X the 1st and 2nd args. of  f
apply(X,1,function(z){f(z[3],z[5])})
```

Apply Tricks: Columns

- Apply the same function `FUN` to every column of an array or dataframe `X` `apply(X,2,FUN)`

```
apply(cats[,2:3],2,median)
```

```
##  Bwt  Hwt
##  2.7 10.1
```

- Same notes as applying across rows

Apply Tricks: Multiple Vectors or Columns

- Given: function `f` which takes 2+ arguments; vectors `x`, `y`, `... z`
- Wanted; `f(x[1],y[1],..., z[1]), f(x[2],y[2],...,z[2])`, etc.
- Solution:

```
mapply(FUN=f,x,y,z)
```

- Will recycle the vectors to the length of the longest if needed
- Often very useful when the vectors are columns, not necessarily from the same object

Transformations

> You go to analysis with the data you have, not the data you want.

The variables in the data are often either not what's most relevant to the analysis, or they're not arranged conveniently, or both
Satisfying model assumptions is a big issue here

$\therefore$ often want to *transform* the data to make it closer to the data we wish we had to start with

Lossless transformations: the original data could be recovered exactly
(at least in principle; function is invertible; same σ-algebra)

Lossy transformations irreversibly destroy some information
(function is not invertible; new σ-algebra is coarser)

Lossless vs. Lossy

Many common transformations are lossless

Many useful transformations are lossy, sometimes very lossy

Because you're documenting your transformations in commented code
yes?
and kept a safe copy of the original data on the disk
yes?
and your disk is backed up regularly
YES?!?
you can use even very lossy transformations without fear

Some Common Transformations of Numerical Data

- `log`: Because $Y = f(X)g(Z) \Leftrightarrow \log Y = \log f(X) + \log g(X)$, taking logs lets us use linear or additive models when the real relationship is multiplicative
  - How would you take the `log` of a whole column?

Numerical Transformations (cont'd.)

- Z-scores, centering and scaling:

```
head(scale(cats[,-1],center=TRUE,scale=TRUE))
```

```
##      Bwt     Hwt
## 1 -1.491 -1.4912
## 2 -1.491 -1.3269
## 3 -1.491 -0.4644
## 4 -1.285 -1.4091
## 5 -1.285 -1.3680
## 6 -1.285 -1.2448
```

- `center=TRUE` ⇒ subtract the mean; alternately, `FALSE` or a vector
- `scale=TRUE` ⇒ divide by standard deviation, after centering; same options
  - Defaults in `scale` produce "Z-scores"

Numerical Transformations (cont'd.)

- Successive differences: `diff(x)`; differences between `x[t]` and `x[t-k]`, `diff(x,lag=k)`
  - Vectorizes over columns of a matrix
- Cumulative totals etc.: `cumsum`, `cumprod`, `cummax`, `cummin`
  - Exercise: write `cummean`
- Rolling means: `rollmean` from the `zoo` package; see Recipe 14.10 in *The R Cookbook*
  - See also Recipe 14.12 on `rollapply`

Numerical Transformations (cont'd.)

- Magnitudes to ranks: `rank(x)` outputs the **rank** of each element of `x` within the vector, 1 being the smallest:

```
head(cats$Hwt)
```

```
## [1] 7.0 7.4 9.5 7.2 7.3 7.6
```

```
head(rank(cats$Hwt))
```

```
## [1]  4.0 11.0 50.5  6.5  9.0 12.5
```

Numerical Transformations (cont'd.)

- "Para-normal" values: Based on the percentile, where would this be if it were Gaussian/normal?

```
qnorm(ecdf(x)(x),mean=100,sd=15)
```

- Obviously nothing magic about using `qnorm` there
- This is how IQ tests are scored; raw scores are highly skewed and don't follow bell curves at all
- "Gaussian copula" = run this trick on two or more variables and then measure the correlations

name due to L. Wasserman

Numerical Transformations (cont'd.)

- Extracting deviations from a trend
  - Calculate the predicted value per trend
  - Take the difference

```
gdp_trend <- gdp[1]*exp(growth.rate*(0:length(gdp)-1))
gdp_vs_trend <- gdp/gdp_rend
```

- Use `residuals` when the trend is a regression model:

```
head(residuals(lm(Hwt ~ Bwt, data=cats)))
```

```
##       1       2       3       4       5       6
## -0.7115 -0.3115  1.7885 -0.9149 -0.8149 -0.5149
```

Summarizing Subsets

- `aggregate` takes a dataframe, a *list* containing the variable(s) to group the rows **by**, and a *scalar* -valued summarizing function:

```
aggregate(cats[,-1],by=cats[1],mean)
```

```
##   Sex  Bwt    Hwt
## 1   F 2.36  9.202
## 2   M 2.90 11.323
```

Note: No comma in `cats[1]`; treating dataframe as a list of vectors - Each vector in the `by` list must be as long as the number of rows of the data

Summarizing Subsets (cont'd.)

- `aggregate` doesn't work on vectors, but it has a cousin, `tapply`:

```
tapply(cats$Hwt,INDEX=cats$Sex,max)
```

```
##    F    M
## 13.0 20.5
```

- tapply can return more than just a scalar value:

```
tapply(cats$Hwt,cats$Sex,summary)
```

```
## $F
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    6.30    8.35    9.10    9.20   10.10   13.00
##
## $M
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     6.5     9.4    11.4    11.3    12.8    20.5
```

Summarizing Subsets (cont'd.)

More complicated actions on subsets usually need the split/apply pattern, which we'll get to in a few weeks

Re-Organizing

- Even if the numbers (or strings, etc.) are fine, they may not be arranged very conveniently
- Lots of data manipulation involves re-arrangement:
  - sorting arrays and dataframes by certain columns
  - exchanging rows and columns
  - merging dataframes
  - Turning short, wide dataframes into long, narrow ones, and vice versa

Re-Ordering

`order` takes in a vector, and returns the vector of indices which would put it in order (increasing by default) - Use the `decreasing=TRUE` option to change that - Output of `order` can be saved to re-order multiple dataframes the same way

order (cont'd.)

```r
head(cats,4)
```

```
##   Sex Bwt Hwt
## 1   F 2.0 7.0
## 2   F 2.0 7.4
## 3   F 2.0 9.5
## 4   F 2.1 7.2
```

```r
head(order(cats$Hwt))
```

```
## [1] 31 48 49  1 13  4
```

```r
head(cats[order(cats$Hwt),],4)
```

```
##    Sex Bwt Hwt
## 31   F 2.4 6.3
## 48   M 2.0 6.5
## 49   M 2.0 6.5
## 1    F 2.0 7.0
```

Related to order

- `rank(x)` does *not* deliver the same thing as `order(x)`!
- `sort` returns the sorted vector, not the ordering

```
head(sort(cats$Hwt))
```

```
## [1] 6.3 6.5 6.5 7.0 7.1 7.2
```

- To just get the index of the smallest or largest element, use `which.min` or `which.max`

```
which.min(cats$Hwt) == order(cats$Hwt)[1]
```

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

Flipping Arrays

- To transpose, converting rows to columns, use `t(x)`
  - Use cautiously on dataframes!
- Use `aperm` similarly for higher-dimensional arrays

Merging Dataframees

You have two dataframes, say `movies.info` and `movies.biz`, and you want to combine them into one dataframe, say `movies`

- Simplest case: the dataframes have exactly the same number of rows, that the rows represent exactly the same units, and you want all columns from both

```
movies <- data.frame(movies.info, movies.biz)
```

Merging Dataframes (cont'd.)

- Next best case: you know that the two dataframes have the same rows, but you only want certain columns from each

```
movies <- data.frame(year=movies.info$year,
  avg_rating=movies.info$avg_rating,
  num_rates=movies.info$num_raters,
  genre=movies.info$genre,
  gross=movies.biz$gross)
```

Merging Dataframes (cont'd.)

- Next best case: same number of rows but in different order
  - Put one of them in the same order as the other
  - Use `merge`
- Worse cases: different numbers of rows...
  - Cleverer re-ordering tricks
  - Use `merge`

An Example That Is Not Part of the Midterm

Claim: People in larger cities travel more

More precise claim: miles driven per person per day increases with the area of the city

Example of Merging (cont'd.)

Distance driven, and city population: table HM-71 in the 2011 "Highway Statistics Series" [http://www.fhwa.dot.gov/policyinformation/statistics/2011/hm71.cfm]

```
fha <- read.csv("fha.csv",na.strings="NA",
                colClasses=c("character","double","double","double"))
nrow(fha)
```

```
## [1] 498
```

```
colnames(fha)
```

```
## [1] "City"                  "Population"            "Miles.of.Road"
## [4] "Daily.Miles.Traveled"
```

Example of Merging (cont'd.)

Area and population of "urbanized areas": [http://www2.census.gov/geo/ua/ua_list_all.txt]

```
ua <- read.csv("ua.txt",sep=";")
nrow(ua)
```

```
## [1] 3598
```

```
colnames(ua)
```

```
##  [1] "UACE"          "NAME"          "POP"           "HU"
##  [5] "AREALAND"      "AREALANDSQMI"  "AREAWATER"     "AREAWATERSQMI"
##  [9] "POPDEN"        "LSADC"
```

Example of Merging (cont'd.)

This isn't a simple case, because:

1. $\approx$ 500 cities vs. $\approx$ 4000 "urbanized areas"
2. `fha` orders cities by population, `ua` is alphabetical by name
3. Both have place-names, but those don't always agree
4. Not even common names for the shared columns

But both use the same Census figures for population, and it turns out every settlement (in the top 498) has a unique Census population:

```
length(unique(fha$Population)) == nrow(fha)
```

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

```
identical(fha$Population,sort(ua$POP[1:nrow(fha)],decreasing=TRUE))
```

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

Example of Merging (cont'd.)

Option 1: re-order the 2nd table by population

```
ua <- ua[order(ua$POP,decreasing=TRUE),]
df1 <- data.frame(fha, area=ua$AREALANDSQMI[1:nrow(fha)])
# Neaten up names
colnames(df1) <- c("City","Population","Roads","Mileage","Area")
nrow(df1)
```

```
## [1] 498
```

```
head(df1)
```

```
##                                  City Population Roads Mileage Area
## 1         New York--Newark, NY--NJ--CT   18351295 43893  286101 3450
## 2 Los Angeles--Long Beach--Anaheim, CA   12150996 24877  270807 1736
## 3                     Chicago, IL--IN    8608208 25905  172708 2443
## 4                           Miami, FL    5502379 15641  125899 1239
## 5         Philadelphia, PA--NJ--DE--MD    5441567 19867   99190 1981
## 6     Dallas--Fort Worth--Arlington, TX    5121892 21610  125389 1779
```

Example of Merging (cont'd.)

Option 2: Use the `merge` function

```r
df2 <- merge(x=fha,y=ua,
            by.x="Population",by.y="POP")
nrow(df2)
```

```
## [1] 498
```

```r
tail(df2,3)
```

```
##     Population                                City Miles.of.Road
## 496    8608208                      Chicago, IL--IN         25905
## 497   12150996 Los Angeles--Long Beach--Anaheim, CA         24877
## 498   18351295            New York--Newark, NY--NJ--CT         43893
##     Daily.Miles.Traveled  UACE                                 NAME
## 496               172708 16264                      Chicago, IL--IN
## 497               270807 51445 Los Angeles--Long Beach--Anaheim, CA
## 498               286101 63217          New York--Newark, NY--NJ--CT
##          HU  AREALAND AREALANDSQMI AREAWATER AREAWATERSQMI POPDEN LSADC
## 496 3459257 6.327e+09         2443 105649916         40.79   3524    75
## 497 4217448 4.496e+09         1736  61141327         23.61   6999    75
## 498 7263095 8.936e+09         3450 533176599        205.86   5319    75
```

merge

- `by.x` and `by.y` say which columns need to match to do a merge
  - Default: merge on all columns with shared names
- New dataframe has *all* the columns of *both* dataframes
  - Here, should really delete the ones we don't need and tidy `colnames`
- If you know databases, then `merge` is doing a `JOIN`
  - If you don't know what that means, wait until November

Example of Merging (cont'd.)

You'd think merging on names would be easy…

```r
df2.1 <- merge(x=fha,y=ua,by.x="City", by.y="NAME")
nrow(df2.1)
```

```
## [1] 492
```

Example of Merging (cont'd.)

We can force unmatched rows of either dataframe to be included, with NA values as appropriate:

```
df2.2 <- merge(x=fha,y=ua,by.x="City",by.y="NAME",all.x=TRUE)
nrow(df2.2)
```

```
## [1] 498
```

Database speak: takes us from a "natural join" to a "left outer join"

Example of Merging (cont'd.)

Where are the mis-matches?

```
df2.2$City[is.na(df2.2$POP)]
```

```
## [1] "Aguadilla--Isabela--San Sebastián, PR"
## [2] "Danville, VA - NC"
## [3] "Florida--Imbéry--Barceloneta, PR"
## [4] "Juana Díaz, PR"
## [5] "Mayagüez, PR"
## [6] "San Germán--Cabo Rojo--Sabana Grande, PR"
```

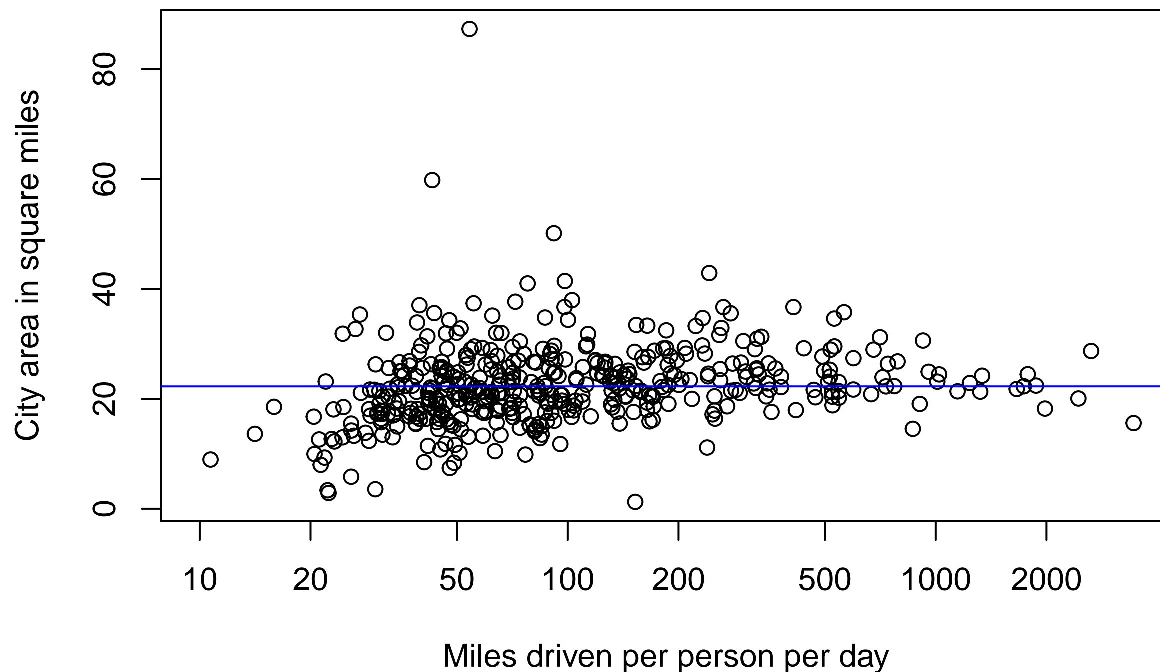
On investigation, `fha.csv` and `ua.txt` use 2 different encodings for accent characters, and one writes things like `VA -- NC` and the other says `VA--NC`

Using order+data.frame vs. merge

- Re-ordering is easier to grasp; `merge` takes some learning
- Re-ordering is simplest when there's only one column to merge on; `merge` handles many columns
- Re-orderng is simplest when the dataframes are the same size; `merge` handles

So, Do Bigger Cities Mean More Driving?

```
# Convert 1,000s of miles to miles
df1$Mileage <- 1000*df1$Mileage
# Plot daily miles per person vs. area
plot(Mileage/Population ~ Area, data=df1, log="x",
     xlab="Miles driven per person per day",
     ylab="City area in square miles")
# Impressively flat regression line
abline(lm(Mileage/Population~Area,data=df1),col="blue")
```



Take-Aways

- Boolean vectors and vectors of indices to access selected parts of the data
- `apply` and friends for doing the same thing to all parts of the data
- Numerical transformations
- Re-ordering dataframes
- Merging dataframes with `merge`

Bonus Topic: Reshaping

- Common to have data where some variables identify units, and others are measurements
- **Wide** form: columns for ID variables plus 1 column per measurement
  - Good for things like correlating measurements, or running regressions
- **Narrow** form: columns for ID variables, plus 1 column identifying measurement, plus 1 column giving value
  - Good for summarizing, subsetting

Often want to convert from wide to narrow, or change what's ID and what's measure

reshape2

- Base R has `reshape` function but it's tricky
- `reshape2` package simplifies lots of common uses

- `melt` turns a wide dataframe into a narrow one
- `dcast` turns a narrow dataframe into a wide one
    - `acast` turns a narrow dataframe into a wide array

Reshaping Example

- `snoqualmie.csv` has precipitation every day in Snoqualmie, WA for 36 years (1948–1983)
- One row per year, one column per day, units of 1/100 inch
  From P. Guttorp, *Stochastic Modeling of Scientific Data* (London: Chapman and Hall, 1995)

```
snoq <- read.csv("snoqualmie.csv",header=FALSE)
colnames(snoq) <- 1:366
snoq[1:3,1:6]
```

```
##     1   2  3  4  5   6
## 1 136 100 16 80 10  66
## 2  17  14  0  0  1  11
## 3   1  35 13 13 18 122
```

```
snoq$year <- 1948:1983
```

Reshaping Example (cont'd.)

```
## Loading required package: reshape2
```

```
snoq.melt <- melt(snoq,id.vars="year",
                  variable.name="day",value.name="precip")
head(snoq.melt)
```

```
##   year day precip
## 1 1948   1    136
## 2 1949   1     17
## 3 1950   1      1
## 4 1951   1     34
## 5 1952   1      0
## 6 1953   1      2
```

Reshaping Example (cont'd.)

Being sorted by day of the year and then by year is a bit odd

```
snoq.melt.chron <- snoq.melt[order(snoq.melt$year,snoq.melt$day),]
head(snoq.melt.chron)
```

```
##     year day precip
## 1   1948   1    136
## 37  1948   2    100
## 73  1948   3     16
## 109 1948   4     80
## 145 1948   5     10
## 181 1948   6     66
```

Reshaping Example (cont'd.)

Most years have 365 days so some missing values:

```
sum(is.na(snoq.melt.chron$precip[snoq.melt.chron$day==366]))
```

```
## [1] 27
```

Tidy with `na.omit`:

```
snoq.melt.chron <- na.omit(snoq.melt.chron)
```

Reshaping Example (cont'd.)

Today's precipitation vs. next day's:

```
snoq.pairs <- data.frame(snoq.melt.chron[-nrow(snoq.melt.chron),],
                         precip.next=snoq.melt.chron$precip[-1])
head(snoq.pairs)
```

```
##     year day precip precip.next
## 1   1948   1    136         100
## 37  1948   2    100          16
## 73  1948   3     16          80
## 109 1948   4     80          10
## 145 1948   5     10          66
## 181 1948   6     66          88
```

Reshaping Example (cont'd.)

- `dcast` turns back into wide form, with a formula of IDs ~ measures

```
snoq.recast <- dcast(snoq.melt,year~...)
```

```
## Using precip as value column: use value.var to override.
```

```
dim(snoq.recast)
```

```
## [1]  36 367
```

```
snoq.recast[1:4,1:4]
```

```
##   year   1   2  3
## 1 1948 136 100 16
## 2 1949  17  14  0
## 3 1950   1  35 13
## 4 1951  34 183 11
```

- `acast` casts into an array rather than a dataframe

Reshaping (cont'd.)

- The formula could also specify multiple ID variables (including original measure variables), different measure variables (including original ID variables)…
- Also possible to apply functions to aggregates which all have the same IDs, select subsets of the data, etc.
- Strongly recommended reading:

  Hadley Wickham, "Reshaping Data with the reshape Package", *Journal of Statistical Software* **21** (2007): 12, [http://www.jstatsoft.org/v21/i12]