Lecture 9: Transforming and Reshaping Data

Statistical Computing, 36-350

Monday October 12, 2015

In previous episodes

- Accessing vectors, arrays, and data frames
- Applying a function across vectors, arrays, data frames
- Extracting data values from 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

Access tricks

How to get the positions in a vector / columns in a matrix / rows in a data frame, matching some condition?

- Vector of Boolean indicators

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

- Recall `cats$Sex=="M"` is a Boolean vector, as long as `cats$Sex`

- Vector of index numbers

```
cats.subset = cats[sample(1:nrow(cats),size=nrow(cats)/2),]
head(cats.subset)
```

```
##     Sex Bwt  Hwt
## 3     F 2.0  9.5
## 74    M 2.6  8.3
## 27    F 2.3 10.1
## 8     F 2.1  8.2
## 56    M 2.2  9.6
## 125   M 3.3 15.4
```

```
head(cats.subset[,"Bwt"])
```

```
## [1] 2.0 2.6 2.3 2.1 2.2 3.3
```

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

```
males = cats$Sex=="M"
row.inds = sample(1:nrow(cats),size=nrow(cats)/2)
# (Now go forth and use these to index)
```

- See also `apply()` tricks below

Access tricks: don't do these

- Non-binary, non-integer vectors do not make good indices

```
n = 50; set.seed(0)
movies = data.frame(gross=round(runif(n,1,500)),
                    genre=sample(c("comedy","action","vampire"),n,replace=T),
                    lead=sample(c("Morgan Freeman","Justin Bieber"),n,replace=T))
movies$lead["Justin Bieber"] # Not going to work!
```

```
## [1] <NA>
## Levels: Justin Bieber Morgan Freeman
```

- Loops are a last resort, not a first

```
# Inefficient and clumsy!
gross.bieber = c()
for (i in 1:nrow(movies)) {
  if (movies$lead[i]=="Justin Bieber") {
    gross.bieber = c(gross.bieber, movies$gross[i])
  }
}
gross.bieber
```

```
##  [1] 448 133 454 102 315  32 104  89 359 389 467 134   8 171 242  94 414
## [18] 335 397  55 362 206 411 324 265 239
```

- Much better here just to define a vector of Booleans

```
movies$gross[movies$lead=="Justin Bieber"]       # Much better
```

```
##  [1] 448 133 454 102 315  32 104  89 359 389 467 134   8 171 242  94 414
## [18] 335 397  55 362 206 411 324 265 239
```

```
movies[movies$lead=="Justin Bieber","gross"]     # Equivalent to above
```

```
##  [1] 448 133 454 102 315  32 104  89 359 389 467 134   8 171 242  94 414
## [18] 335 397  55 362 206 411 324 265 239
```

```
mean(movies$gross[movies$lead=="Justin Bieber"]) # Bieber avg
```

```
## [1] 251.8846
```

```
mean(movies$gross[movies$lead!="Justin Bieber"]) # Freeman avg
```

```
## [1] 287.1667
```

Apply tricks, continued

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

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

```
## [1] 50  3
```

- Math functions are vectorized

```
mean(log(movies$gross))
```

```
## [1] 5.352082
```

- Distribution-related functions are vectorized

```
rnorm(n=5,mean=-2:2,sd=0.001*(1:5)) # Check that you understand this!
```

```
## [1] -1.9988990309 -0.9997124570 -0.0003532608  0.9963517265  1.9928120688
```

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

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

```
sd(jackknifed.means)*sqrt((n-1)^2/n)
```

```
## [1] 0.04044222
```

- `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 data frame `X`: `apply(X,1,FUN)`

```
movies$gross[sample(1:nrow(movies),2)] = NA
movies$genre[sample(1:nrow(movies),1)] = NA
movies$lead[sample(1:nrow(movies),1)] = NA
rows.with.NAs = apply(is.na(movies),1,any)
length(rows.with.NAs)
```

```
## [1] 50
```

```
sum(rows.with.NAs)
```

```
## [1] 4
```

- `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 this work

```
# Let's rewrite the books and pretend Bieber grossed more $$...
gross.fake = apply(movies,1, function(r) {
  return(ifelse(r[3]=="Justin Bieber",
                1.1*as.numeric(r[1]),
                0.9*as.numeric(r[1])))})
movies$gross = gross.fake
mean(movies$gross[movies$lead=="Justin Bieber"],na.rm=T) # New Bieber avg
```

```
## [1] 286.748
```

```
mean(movies$gross[movies$lead!="Justin Bieber"],na.rm=T) # New Freeman avg
```

```
## [1] 263.4955
```

(Why did we have to do such trickery? Turns out apply is mostly mean to work with matrices, and will cast each row to be a single data type, here a string

Apply tricks, columns

- Apply the same function `FUN()` to every column of an array or data frame `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 arguments

- Given a function `f` which takes 2+ arguments; vectors `x`, `y`, ... `z`, suppose we want `f(x[1],y[1],...,z[1]), f(x[2],y[2],...,z[2])`, etc. How?

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

- Will recycle the vectors to the length of the longest if needed
- Super useful, saves you a for loop

Transformations

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

Hence we 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
- **Lossy** transformations irreversibly destroy some information

Lossless versus lossy

- Many common transformations are lossless
- Many useful transformations are lossy, sometimes very lossy
- But that's OK, because
  - You're documenting your transformations in commented code (right?)
  - And kept a safe copy of the original data on the disk (right??)
  - And your disk is backed up regularly (right???)
- So you can use even very lossy transformations without fear

Log transforms

Because

$$Y = f(X)g(Z) \iff \log Y = \log f(X) + \log g(X),$$

taking logs lets us use (say) linear models when the real relationship is multiplicative

Variance stabilizing transforms

Sometimes the variance of our random variables might grow with the mean, and this would not be good for (say) a linear model

E.g., Poisson distribution: if $Y \sim \text{Poisson}(\lambda)$, then $\mathbb{E}(Y) = \lambda$ and $\text{Var}(Y) = \lambda$. To stabilize the variance, we transform to $Z = \sqrt{Y}$, which has

$$\mathbb{E}(Z) \approx \sqrt{\lambda}, \quad \text{and} \quad \text{Var}(Z) \approx 1/4$$

(This is a result of what is called the **delta method**, which you can view as a first-order Taylor series approximation)

```
lambda = (1:5)^2
num = 10000
Y = matrix(rpois(num*length(lambda),
                 lambda=rep(lambda,each=num)),
           nrow=num)
apply(Y,2,mean)
```

```
## [1]  1.0047  4.0016  8.9696 16.0497 25.1090
```

```
apply(Y,2,var)
```

```
## [1]  1.010179  3.943792  8.871563 15.980028 25.595079
```

```
apply(sqrt(Y),2,mean)
```

```
## [1] 0.7731338 1.9229928 2.9512973 3.9741548 4.9851141
```

```
apply(sqrt(Y),2,var)
```

```
## [1] 0.4070049 0.3037290 0.2594699 0.2558191 0.2576628
```

Some common numerical transforms

- Centering and scaling: `scale(x,center=T,scale=F)` subtracts the mean from each column of `x`; `scale(x,center=F,scale=T)` divides each column by its sd; `scale(x,center=T,scale=T)` does both
- Successive differences: `diff()`; lags are possible; higher-order differences are possible; vectorizes over columns of a matrix
- Cumulative functions: `cumsum()`, `cumprod()`, `cummax()`, `cummin()`
- Rolling means: `rollmean()` from the `zoo` package; see Recipe 14.10 in the R Cookbook

Ranks

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

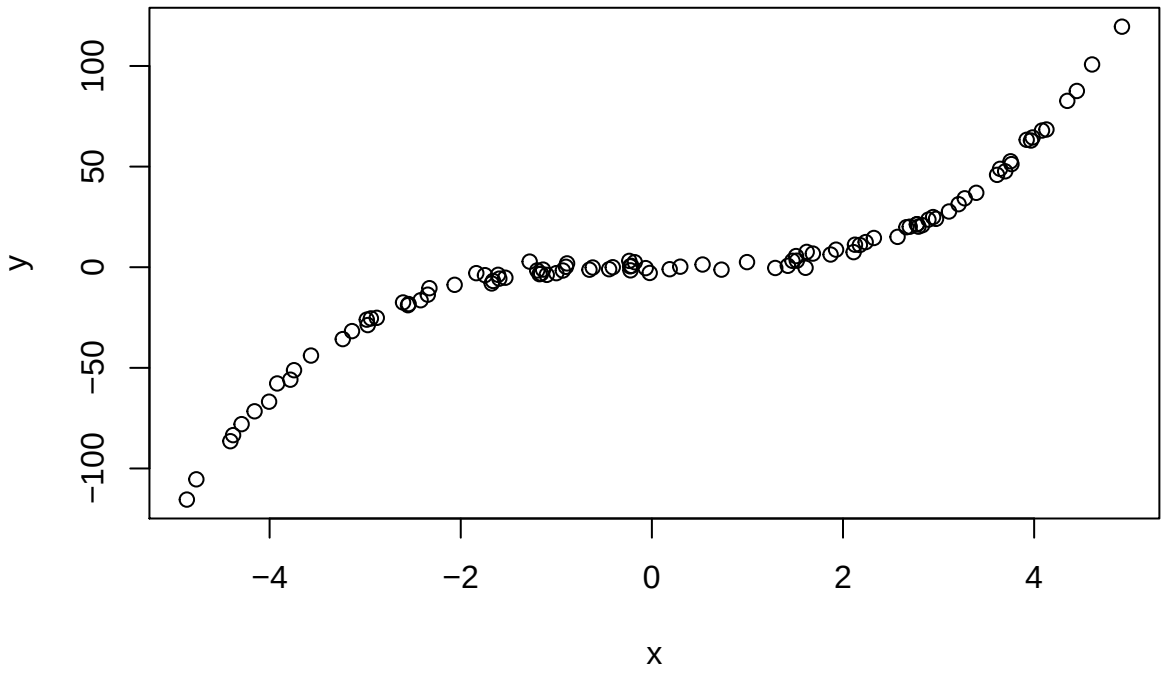
```
n = 100; set.seed(0)
x = runif(n,-5,5)
head(rank(x))
```

```
## [1] 93 22 32 56 95 14
```

```
y = x^3 + rnorm(n,sd=2)
cor(x,y)
```

```
## [1] 0.903845
```

```
plot(x,y)
```




```
cor(rank(x),rank(y)) # Called Spearman's correlation
```

```
## [1] 0.9830423
```

Summarizing subsets

- Run a particular function over various groups of your data vector `X`, with `tapply(X,INDEX,FUN)`:

```
tapply(cats$Hwt,cats$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.
##   6.300   8.350   9.100   9.202  10.100  13.000
##
## $M
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    6.50    9.40   11.40   11.32   12.80   20.50
```

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 data frames by certain columns
  - Exchanging rows and columns
  - Merging data frames
  - Turning short, wide data frames 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 increasing order
- Use the `decreasing=TRUE` option to get decreasing order
- Output of `order` can be saved to re-order multiple data frames the same way

```
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
```

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

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

```
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
```

- `rank()` and `order()` are effectively inverses of each other
- `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 data frames! (Try it with `movies`)
- Use `aperm` similarly for higher-dimensional arrays

Merging data frames

Suppose you have two data frames `X`, `Y`, and you want to combine them

- Simplest case: the data frames have exactly the same number of rows, that the rows represent exactly the same units, and you want all columns from both; just use, `data.frame(X,Y)`
- Next best case: you know that the two data frames have the same rows, but you only want certain columns from each; just use, e.g., `data.frame(X$col1,X$col5,Y$favcol)`
- Next best case: same number of rows but in different order; put one of them in same order as the other, with `order()`. Alternatively, use `merge()`
- Worse cases: different numbers of rows ... hard to line up rows ...

Example: big city drivers

Seems reasonable that people in larger cities (larger areas) would drive more. Is this true?

Distance driven, and city population: [http://www.fhwa.dot.gov/policyinformation/statistics/2011/hm71.cfm]

```
fha = read.csv("http://www.stat.cmu.edu/~ryantibs/statcomp/lectures/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"
```

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

```
ua = read.csv("http://www.stat.cmu.edu/~ryantibs/statcomp/lectures/ua.txt",sep=";")
nrow(ua)
```

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

```
colnames(ua)
```

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

This isn't a simple case, because:

- ≈ 500 cities vs ≈ 4000 "urbanized areas"
- `fha` orders cities by population, `ua` is alphabetical by name
- Both have place-names, but those don't always agree
- 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
```

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

```
## [1] 0
```

First way to merge

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 3450.20
## 2 Los Angeles--Long Beach--Anaheim, CA   12150996 24877  270807 1736.02
## 3                      Chicago, IL--IN    8608208 25905  172708 2442.75
## 4                           Miami, FL    5502379 15641  125899 1238.61
## 5         Philadelphia, PA--NJ--DE--MD    5441567 19867   99190 1981.37
## 6    Dallas--Fort Worth--Arlington, TX    5121892 21610  125389 1779.13
```

Second way to merge

Option 2: Use the `merge()` function

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

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

```
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 6326686332      2442.75 105649916         40.79 3524.0    75
## 497 4217448 4496266014      1736.02  61141327         23.61 6999.3    75
## 498 7263095 8935981360      3450.20 533176599        205.86 5318.9    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 data frame has all the columns of both data frames
  - Here, should really delete the ones we don't need and tidy the remaining names
- If you know databases, then `merge()` is doing a `JOIN`
  - If you don't know what that means, wait a few weeks ...

Merging on names doesn't work here

```
df3 = merge(x=fha,y=ua,by.x="City", by.y="NAME")
nrow(df3)
```

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

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

```
df4 = merge(x=fha,y=ua,by.x="City",by.y="NAME",all.x=TRUE)
nrow(df4)
```

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

Where are the mis-matches?

```
df4$City[is.na(df4$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"
```

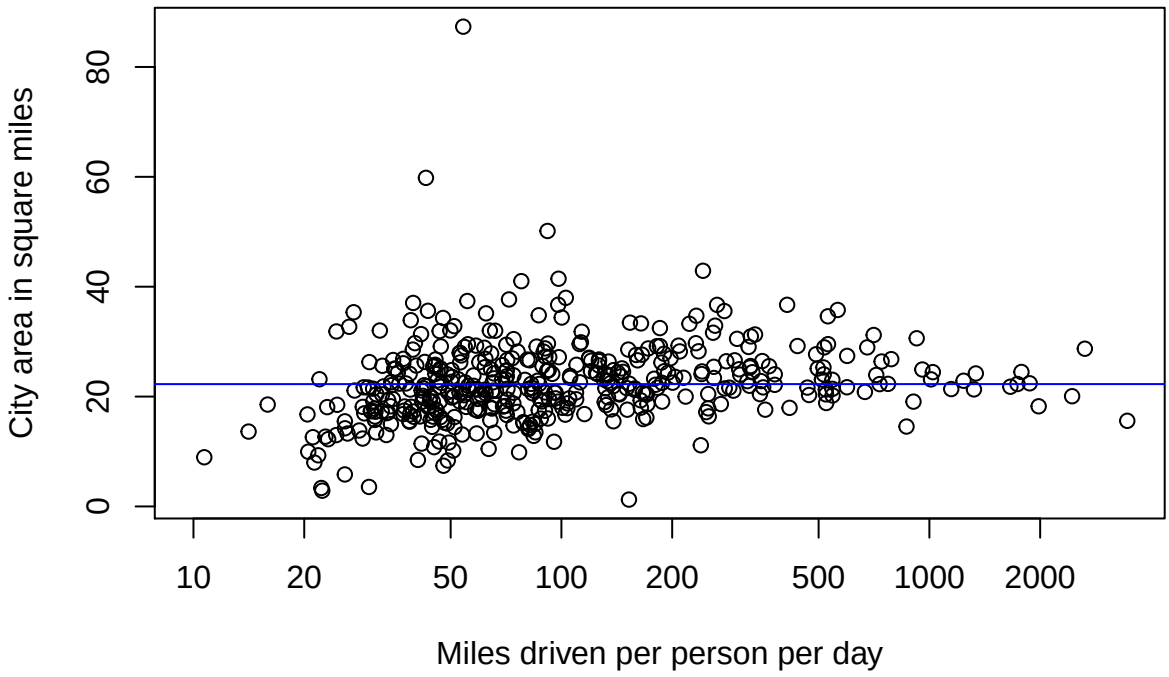
(On investigation, `fha.csv` and `ua.txt` use 2 different encodings for accent characters, and one writes things like `VA -- NC` and the other says `VA--NC`)

Using `order()` and manual tricks versus `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 data frames are the same size; `merge` handles different sizes

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")
```



Summary

- 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 data frames
- Merging data frames with `order()` (basic way), `merge()` (more advanced)