Lecture 11: Data Transformations

STAT GR5206

Statistical Computing & Introduction to Data Science

Cynthia Rush
Columbia University

November 17, 2017

Course Notes

- Homework due November 27. One more homework after this one.
- No lab next week.
- Final Friday, December 15, 1:10pm - 4:00pm. (Location TBD.)
- Can't stick around after class today.

Mostly Review: Selective Access and the apply() Family

Selective Access

First up, we review how to selectively access parts of the data.

Goal: Find the rows in a dataframe matching some condition.

- Use logicals: create a vector of Boolean (logical) values.
- Use indices: create a vector of index numbers.

Selective Access

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

# One way to find heart weight for male cats
head(cats$Hwt[cats$Sex=="M"])
head(cats[cats$Sex=="M","Hwt"])

cats.subset <- sample(1:nrow(cats), size = nrow(cats)/2)
head(cats.subset)
new.cats <- cats[cats.subset,]
head(new.cats, 3)

# In both cases, can save and re-use the values.
males   <- cats$Sex == "M"
row.ind <- sample(1:nrow(cats), size = nrow(cats)/2)
```

Selective Access

- Another way of accessing a subset of the rows, sometimes easier with data frames, is through `subset()`.
- Using `subset()`, we can just use the column names directly, i.e., no need for writing `cats$Sex`, can just use `Sex`;

```
boy.cats.1 <- subset(cats, Sex == "M")

# Get same thing by extracting the rows manually
boy.cats.2 <- cats[cats$Sex=="M", ]
all(boy.cats.1 == boy.cats.2)
```

Selective Access: Don't Do This

- Non-binary, non-integer vectors can't be used to index data.

```
# Matrix of states data, 50 states x 9 variables
states <- data.frame(state.x77, Region = state.region)
head(states, 3)
states$Income["South"] # doesn't work
```

- Loops are a last resort, not a first.

```
# This is inefficient and clumsy
income.south <- c()
for (i in 1:nrow(states)) {
  if (states$Region[i] == "South") {
    income.south <- c(income.south, states$Income[i])
  }
}
income.south
```

- Much better to just define a vector of logical values.

```
states$Income[states$Region == "South"]
states[states$Region == "South", "Income"]
```

Review: The `apply()` Family

Vectorized Functions

- 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
```

- If the function doesn't vectorize, or it doesn't quite do what you want, turn to the `apply()` family of functions.

Review: The `apply()` Family

- `R` offers a family of `apply()` functions, which allow you to apply a function across different chunks of data.
- Offers an alternative to explicit iteration using `for()` loop.
- Almost always simpler and faster than a `for()` loop.

Below is a summary.

- `apply()`: apply a function to rows or columns of a matrix or data frame.
- `lapply()`: apply a function to elements of a list or vector.
- `sapply()`: same as the above, but simplify the output (if possible).
- `tapply()`: apply a function to levels of a factor vector.
- `mapply()`: apply a function to multiple vector arguments.

Check Yourself

Tasks: Don't Use `apply()`

- How many states have at least 150 days of frost per year? Which ones are they? (Hint: you should only need one line of code to answer each question.)
- For each of the 8 numeric variables in `states`, what is the average? For each of the variables, how many states have values above the average? (Hint: You may want to use `colSums()` here.)

Review: The apply() Family

apply() for Matrices and Dataframes

`apply(X, MARGIN, FUN)` applies the same function `FUN` to every row (`MARGIN = 1`) or column (`MARGIN = 2`) of an array or dataframe `X`.

```
# Maximum entry in each column
apply(states[,1:8], MARGIN = 2, FUN = max)
apply(states[,1:8], MARGIN = 2, FUN = which.max)
rownames(states)[apply(states[,1:8], MARGIN = 2, FUN = which.max
```

```
# Summary of each col, get back matrix!
apply(states[,1:5], MARGIN = 2, FUN = summary)
```

Review: The apply() Family

- `apply()` tries to return a vector or a matrix; will return a list if it can't.
- `apply()` assumes `FUN` will work on a row (`MARGIN = 1`) or column (`MARGIN = 2`) of `X`; might need to write a little adapter function to make that true.

```
# Rewrite the books so the Northeast gets less frost
frow <- function (r) {
  val <- as.numeric(r[7])
  return(ifelse(r[9] == "Northeast", 0.5*val, val))
}
frost.fake <- apply(states, 1, frow)

mean(states$Frost[states$Region == "Northeast"])
mean(frost.fake[states$Region == "Northeast"])
```

Review: The `apply()` Family

Why did we have to use `as.numeric()` in the `apply()` call above?

```
frow <- function (r) {
  val <- as.numeric(r[7])
  return(ifelse(r[9] == "Northeast", 0.5*val, val))
}
```

Since `apply()` is meant to work with matrices, it casts each row to be a single data type – here a string.

Review: The apply() Family

Sometimes we want to use a function over rows or columns of a matrix, that takes extra arguments (besides the row or column itself).

Pass additional arguments as inputs to `apply()`, as in:

```
apply(x, MARGIN = 1, FUN = my.fun, extra.arg.1, extra.arg.2)
```

for two extra arguments to be passed to `my.fun()`.

```
# Goal: find indices of biggest 3 entries of v,
# Return: corresponding elements of names.v
top.3.names = function(v, names.v) {
  names.v[order(v, decreasing=TRUE)[1:3]]
}

# Run the function on each column of states. Note: here
# v is be a column, and names.v is the state names
apply(states[, 1:7], MARGIN = 2, FUN = top.3.names,
      names.v = rownames(states))
```

What's the return argument?

What kind of data type will `apply()` give us? Depends on what function we pass. Here's a summary using `FUN = my.fun()`:

- If `my.fun()` returns a single value, then `apply()` returns a vector.
- If `my.fun()` returns k values, then `apply()` returns a matrix with k rows (note: this is true regardless of whether `MARGIN = 1` or `MARGIN = 2`).
- If `my.fun()` returns different length output for different inputs, then `apply()` returns a list.
- If `my.fun()` returns a list, then `apply()` returns a list.

Check Yourself

Tasks

- How does `Frost` correlate with the others variables? Write a function `cor.v1.v2()` that takes two inputs: `v1`, a numeric vector; and `v2`, another numeric vector, whose default value is `states[, "Frost"]`. Its output should be the correlation of `v1` and `v2`. (Hint: Use `cor()`.)
- Using `apply()` and `cor.v1.v2()`, calculate the correlation between each one of the 8 numeric variables in the `states` matrix and the `Frost` variable.
- Can you do accomplish the above using `cor()` directly and passing additional arguments to `apply()`?

sapply() and lapply() for Vectors

sapply() or lapply() allow you to apply the same function to every element in a list or a vector.

```
# Computes leave-one-out means, also called jackknife means.
mean.less.one <- function(i, vec) {
  return(mean(vec[-i]))
}

my.vec <- states[ ,"Frost"]
n      <- length(my.vec)
# my.vec is an additional argument to mean.omitting.one
my.vec.jack <- lapply(1:n, FUN = mean.less.one, vec = my.vec)

# It's a list, and here are the first 3 elements
head(my.vec.jack, 3)
```

sapply() and lapply() for Vectors

sapply() function works just like lapply(), but tries to **simplify** the return value whenever possible. (lapply() always returns a list.)

```
# my.vec is an additional argument to mean.omitting.one
my.vec.jack <- sapply(1:n, FUN = mean.less.one, vec = my.vec)

# It's a vector, and here are the first 5 elements
head(my.vec.jack)
```

tapply() for Levels of a Factor

The function `tapply()` takes inputs as in:

`tapply(x, INDEX = my.index, FUN = my.fun),`

to apply `my.fun()` to subsets of entries in `x` that share a common level in `my.index`.

```
# Let's avg the Frost variable, within in each region
tapply(states[,"Frost"], INDEX = state.region, FUN = mean)
```

Review: The `apply()` Family

A New One: `mapply()`

- **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**: Multivariate apply

  `mapply(FUN = f, x, y, z)`
- Will recycle the vectors to the length of the longest if needed
- Often very useful and can replace loops.

```
mapply(rep, 1:4, 4:1)
```

Mostly Review: Re-ordering and Merging Dataframes

Re-Organizing Data

- 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 Data

Sometimes it's convenient to reorder our data, say the rows of our data frame (or matrix). Recall:

`order()` takes in a vector, and returns the vector of indices that puts the vector in order (increasing by default).

- Use the `decreasing = TRUE` option to get decreasing order.
- The output of `order` can be saved to re-order all the columns of dataframes simultaneously.

Compare with `rank()` and `sort()`.

Re-Ordering Data

The cats data has its rows ordered by the smallest body weight to the largest in females, and then in males. Suppose we wanted to order by smallest to largest heart weight.

```
head(cats, 3)
hwt.order  <- order(cats$Hwt)   # By increasing heart weight
cats.order <- cats[hwt.order, ] # Reorder rows
head(cats.order, 3)

# Rank vs order vs sort
this.vec <- c(25, 13, 25, 77, 68)
rank(this.vec)
order(this.vec)
this.vec[order(this.vec)]
sort(this.vec)
```

Finding the Maximum

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]
```

Flipping Arrays

- To transpose, converting rows to columns, use `t(x)`.
- Use cautiously on dataframes!

```
t(cats)[, 1:5]
```

Merging Dataframes

Suppose you have two dataframes, `X` and `Y`, and you want to combine them into one dataframe.

- **Simplest case**: the dataframes have exactly the same number of rows, 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 dataframes have the same rows, but you only want certain columns from each; just use, for example, `data.frame(X$col1, X$col5, Y$favcol)`.
- **Next best case**: same number of rows but in different order; put one of them in the same order as the other with `order()`. Alternately, use `merge()`.
- **Worse cases**: different numbers of rows or hard to line up rows; use more clever re-ordering tricks or use `merge()`.

An Example

Claim: People in larger cities drive more.

More precise claim: Miles per person per day increases with city area.

The Data

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

```
fha <- read.csv("fha.csv", na.strings = "NA",
                colClasses = c("character", rep("double", 3)))
nrow(fha)
colnames(fha)
head(fha, 3)
```

Merging Dataframes

The Data

Area and population of "urbanized areas" from `http://www2.census.gov/`.

```
ua <- read.csv("ua.txt", sep = ";")
nrow(ua)
head(ua, 2)
```

An Example

Difficulties in merging the two datasets:

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$Population)) == nrow(fha)
ua.pop.top498 = sort(ua$POP, decreasing = TRUE)[1:nrow(fha)]
max(abs(fha$Population - ua.pop.top498))
```

An Example

Option 1: Reorder area column in `ua` table by population, append to `fha`

```
# Order by population

ua.sort <- ua[order(ua$POP, decreasing = TRUE), ]
area    <- ua.sort$AREALANDSQMI[1:nrow(fha)]
df1     <- data.frame(fha, area)

# Neaten up names

colnames(df1) <- c("City","Population","Roads",
                   "Mileage","Area")
nrow(df1)
head(df1, 3)
```

Merging Dataframes

Option 2: Use the `merge()` function

```
df2 <- merge(x = fha, y = ua, by.x = "Population", by.y = "POP")
nrow(df2)
tail(df2, 2)
```

An Example

The `merge()` function tries to merge two data frames according to common columns, as in:

```
merge(x, y, by.x = "SomeXCol", by.y = "SomeYCol"),
```

to join two data frames `x`, `y`, by matching the columns `SomeXCol` and `SomeYCol`.

- Default (when no `by.x` and `by.y` are specified) is to merge on all columns with shared names.
- Output will be a new data frame that has all the columns of both data frames.
- Should really delete the columns 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, you will by the end of the semester!)

An Example

You'd think merging on names would be easy...

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

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)
```

An Example

Where are the mis-matches?

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

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`

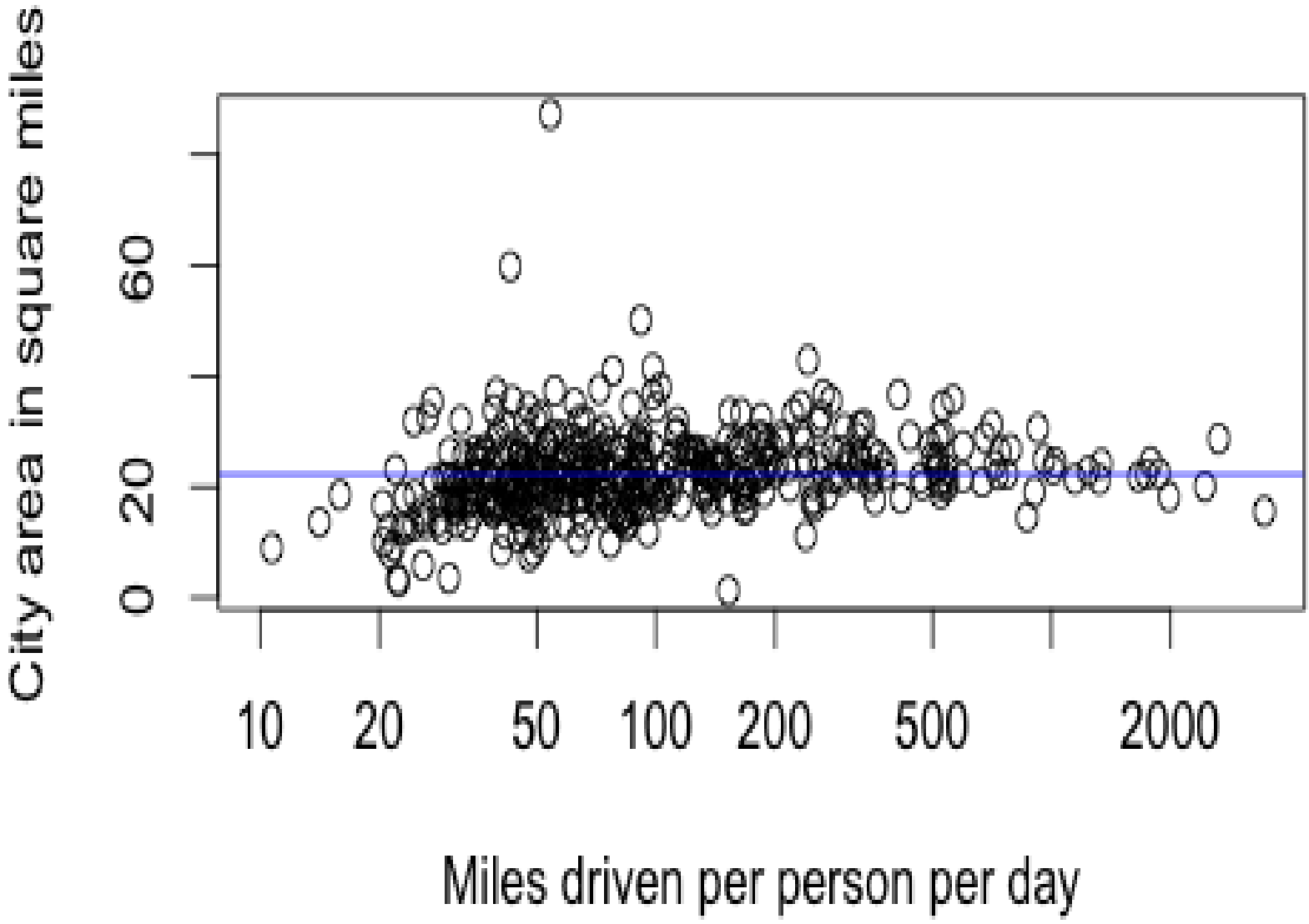
Using `order()` and manual tricks vs. `merge()`

- Reordering is easier to grasp; `merge()` takes some learning
- Reordering is simplest when there's only one column to merge on; `merge()` handles many columns
- Reordering is simplest when the dataframes are the same size; `merge()` handles different sizes automatically

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 (sq. miles)")

# Impressively flat regression line

abline(lm(Mileage/Population ~ Area, data = df1),
       col = "blue")
```

So, Do Bigger Cities Mean More Driving?



Transforming Data

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.
- ∴ often want to **transform** the data to make it closer to the data we wish we had to start with.
- Two types:
  - **Lossless** transformations: the original data could be recovered exactly.
  - **Lossy** transformations irreversibly destroy some information.

Lossless vs. Lossy

- Many common transformations are lossless
- Many useful transformations are lossy, sometimes very lossy
- BUT,
  1. Because you're documenting your transformations in commented code
  2. and kept a safe copy of the original data on the disk
  3. and your disk is backed up regularly

  you can use even very lossy transformations without fear

Some Common Transformations

Z-scores, centering and scaling:

```
head(scale(cats[,-1], center = TRUE, scale = TRUE), 3)
```

- `center = TRUE` $\Rightarrow$ subtracts the mean from each column;
- `scale = TRUE` $\Rightarrow$ divides each column by standard deviation, after centering;
- Defaults in `scale` produce "Z-scores"

Some Common Transformations

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

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

Check Yourself

Tasks

For each of the 8 numeric variables in `states`, we want to calculate the geometric mean. Recall, that the geometric mean of a vector of length n is the product of its entries, all raised to the power of 1/n.

$$\text{For } (x_1, \ldots, x_n), \quad \text{geometric mean } = \left(\prod_{i=1}^{n} x_i \right)^{1/n}.$$

Solve this in two ways:

1. After you transform the data in a clever way, calculate the geometric mean using `colMeans()`. Hint: think about the relationship between sums of logs and logs of products.
2. Write a `geom.mean()` function which takes as input `the.vec` and returns the geometric mean. Then apply this function to each column of the data.

Sometimes we want to split up the rows of a data frame or entries of a vector by levels of a factor.

`split()` by Levels of a Factor

`split(x, f = my.index)` splits a data frame or vector `x` according to levels of `my.index`

```
# Let's split up the states matrix according to region

states.by.reg = split(states, f = states$Region)
class(states.by.reg) # The result is a list
names(states.by.reg) # With 4 elements for the 4 regions
class(states.by.reg[[1]]) # Each element is a data frame
```

Summarizing Subsets

```
# For each region, display first 2 rows of the data frame
lapply(states.by.reg, FUN = head, 2)

# For each region, average the 8 numeric variables
mean.fun <- function(df) {
  apply(df[, 1:8], MARGIN = 2, mean, na.rm = TRUE)
}
lapply(states.by.reg, mean.fun)
```

`aggregate()` by Levels of a Factor

`aggregate(x, by, FUN)` takes a dataframe, `x`, a list containing the variable(s) to group the rows `by`, and a scalar-valued summarizing `FUN`.

```
aggregate(states[,1:8], by = list(states$Region), mean)
```

`aggregate()` by Levels of a Factor

- Each vector in the `by` list must be as long as the number of rows of the data.
- `aggregate()` doesn't work on vectors. There you use `tapply()`.
- More complicated actions on subsets usually need the split/apply pattern, which we'll talk about in a bit.

Check Yourself

Tasks

- Split the rows of `states` by division using the `split()` function, and call the resulting list `states.by.div`, (having length 9, with one element per division).
- Display the first 2 rows of each data frame in the list `states.by.div`.
- Aggregate your data by `Region` and `Division` and summarize the data by finding the mean.

Check Yourself

Tasks

- For each division, compute the median graduate-by-literate percentage (we wrote a `grad.by.lit.median()` function earlier).
- For each division, compute the median HS graduation percentage. Do so using `sapply()` on `states.by.div`, with the `FUN` input defined "on-the-fly", meaning in the function call.