More Data Structures

James M. Flegal

Agenda

- Arrays
- Matrices
- Lists
- Dataframes
- Structures of structures

Arrays

- Many data structures in R are made by adding bells and whistles to vectors, so "vector structures"
- Most useful is **arrays**

```
x <- c(7, 8, 10, 45)
x.arr <- array(x,dim=c(2,2))
x.arr
```

```
##      [,1] [,2]
## [1,]    7   10
## [2,]    8   45
```

- `dim` says how many rows and columns; filled by columns
- Can have $3, 4, \ldots n$ dimensional arrays; `dim` is a length-n vector

Arrays

- Some properties of the array

```r
dim(x.arr)
```

```
## [1] 2 2
```

```r
is.vector(x.arr)
```

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

```r
is.array(x.arr)
```

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

Arrays

```r
typeof(x.arr)
```

```
## [1] "double"
```

```r
str(x.arr)
```

```
##  num [1:2, 1:2] 7 8 10 45
```

```r
attributes(x.arr)
```

```
## $dim
## [1] 2 2
```

- `typeof()` returns the type of the *elements*
- `str()` gives the **structure**: here, a numeric array, with two dimensions, both indexed 1–2, and then the actual numbers
- Exercise: try all these with `x`

Accessing and operating on arrays

- Can access a 2-D array either by pairs of indices or by the underlying vector

```
x.arr[1,2]
```

```
## [1] 10
```

```
x.arr[3]
```

```
## [1] 10
```

Accessing and operating on arrays

- Omitting an index means "all of it"

```
x.arr[c(1:2),2]
```

```
## [1] 10 45
```

```
x.arr[,2]
```

```
## [1] 10 45
```

Functions on arrays

- Using a vector-style function on a vector structure will go down to the underlying vector, *unless* the function is set up to handle arrays specially

```
which(x.arr > 9)
```

```
## [1] 3 4
```

Accessing and operating on arrays

- Many functions *do* preserve array structure

```
y <- -x
y.arr <- array(y,dim=c(2,2))
y.arr + x.arr
```

```
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
```

- Others specifically act on each row or column of the array separately

```
rowSums(x.arr)
```

```
## [1] 17 53
```

- See more of this idea later

Example: Price of houses in PA

- Census data for California and Pennsylvania on housing prices, by Census "tract"

```
calif_penn <- read.csv("http://www.stat.cmu.edu/~cshalizi/uADA/13/hw/01/calif_penn_2011.csv")
penn <- calif_penn[calif_penn[,"STATEFP"]==42,]
coefficients(lm(Median_house_value ~ Median_household_income, data=penn))
```

```
##             (Intercept) Median_household_income
##            -26206.564325                3.651256
```

- Fit a simple linear model, predicting median house price from median household income

Example: Price of houses in PA

- Census tracts 24–425 are Allegheny county
- Tract 24 has a median income of \$14,719; actual median house value is \$34,100 — is that above or below what's predicted?

```
34100 < -26206.564 + 3.651*14719
```

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

- Tract 25 has income \$48,102 and house price \$155,900

```
155900 < -26206.564 + 3.651*48102
```

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

- What about tract 26?

Example: Price of houses in PA

- *Could* just keep plugging in numbers like this, but that's
  - boring and repetitive
  - error-prone (what if I forget to change the median income, or drop a minus sign from the intercept?)
  - obscure if we come back to our work later (what *are* these numbers?)

Use variables and names

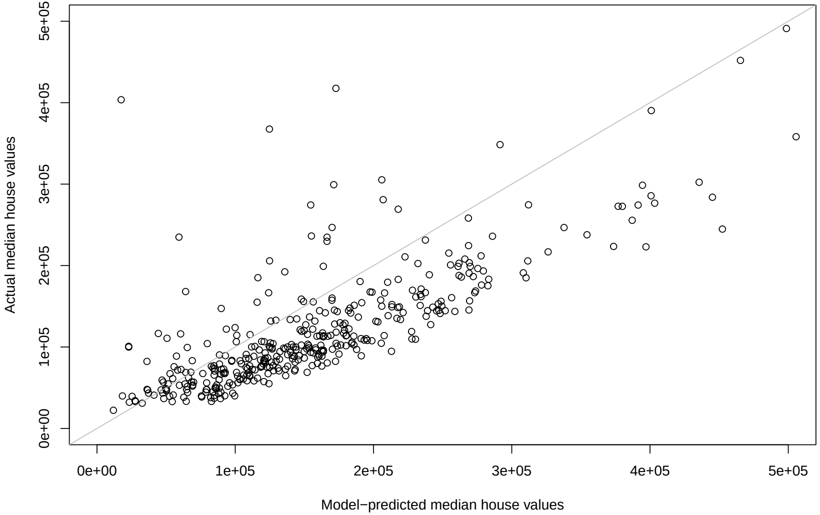
```
penn.coefs <- coefficients(lm(Median_house_value ~ Median_household_income, data=penn))
penn.coefs
```

```
##             (Intercept) Median_household_income
##           -26206.564325                3.651256
```

```
allegheny.rows <- 24:425
allegheny.medinc <- penn[allegheny.rows,"Median_household_income"]
allegheny.values <- penn[allegheny.rows,"Median_house_value"]
allegheny.fitted <- penn.coefs["(Intercept)"]+penn.coefs["Median_household_income"]*allegheny.medinc
```

Use variables and names

```
plot(x=allegheny.fitted, y=allegheny.values, xlab="Model-predicted median house values",
     ylab="Actual median house values", xlim=c(0,5e5),ylim=c(0,5e5))
abline(a=0,b=1,col="grey")
```



Resource allocation example

- Factory makes cars and trucks, using labor and steel
  - a car takes 40 hours of labor and 1 ton of steel
  - a truck takes 60 hours and 3 tons of steel
  - resources: 1600 hours of labor and 70 tons of steel each week

Matrices

- Matrix is a specialization of a 2D array

```
factory <- matrix(c(40,1,60,3),nrow=2)
is.array(factory)
```

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

```
is.matrix(factory)
```

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

- Could also specify `ncol`, and/or `byrow=TRUE` to fill by rows.
- Element-wise operations with the usual arithmetic and comparison operators (e.g., `factory/3`)
- Compare whole matrices with `identical()` or `all.equal()`

Matrix multiplication

- Gets a special operator

```
six.sevens <- matrix(rep(7,6),ncol=3)
six.sevens
```

```
##      [,1] [,2] [,3]
## [1,]    7    7    7
## [2,]    7    7    7
```

```
factory %*% six.sevens # [2x2] * [2x3]
```

```
##      [,1] [,2] [,3]
## [1,]  700  700  700
## [2,]   28   28   28
```

- What happens if you try `six.sevens %*% factory`?

Multiplying matrices and vectors

- Numeric vectors can act like proper vectors

```r
output <- c(10,20)
factory %*% output
```

```
##      [,1]
## [1,] 1600
## [2,]   70
```

```r
output %*% factory
```

```
##      [,1] [,2]
## [1,]  420  660
```

- R silently casts the vector as either a row or a column matrix

Matrix operators

- Transpose

```r
t(factory)
```

```
##      [,1] [,2]
## [1,]   40    1
## [2,]   60    3
```

- Determinant

```r
det(factory)
```

```
## [1] 60
```

The diagonal

- The `diag()` function can extract the diagonal entries of a matrix

```r
diag(factory)
```

```
## [1] 40  3
```

- It can also *change* the diagonal

```r
diag(factory) <- c(35,4)
factory
```

```
##      [,1] [,2]
## [1,]   35   60
## [2,]    1    4
```

- Re-set it for later

```r
diag(factory) <- c(40,3)
```

Creating a diagonal or identity matrix

```r
diag(c(3,4))
```

```
##      [,1] [,2]
## [1,]    3    0
## [2,]    0    4
```

```r
diag(2)
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Inverting a matrix

```r
solve(factory)
```

```
##                [,1]       [,2]
## [1,]  0.05000000 -1.0000000
## [2,] -0.01666667  0.6666667
```

```r
factory %*% solve(factory)
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Why's it called "solve" anyway?

- Solving the linear system $\mathbf{A}\vec{x} = \vec{b}$ for $\vec{x}$

```
available <- c(1600,70)
solve(factory,available)
```

```
## [1] 10 20
```

```
factory %*% solve(factory,available)
```

```
##      [,1]
## [1,] 1600
## [2,]   70
```

Names in matrices

- Can name either rows or columns or both, with `rownames()` and `colnames()`
- Just character vectors, and we use the same function to get and to set their values
- Names help us understand what we're working with
- Can be used to coordinate different objects

Names in matrices

```r
rownames(factory) <- c("labor","steel")
colnames(factory) <- c("cars","trucks")
factory
```

```
##       cars trucks
## labor   40     60
## steel    1      3
```

```r
available <- c(1600,70)
names(available) <- c("labor","steel")
```

Names in matrices

```
output <- c(20,10)
names(output) <- c("trucks","cars")
factory %*% output # But we've got cars and trucks mixed up!
```

```
##       [,1]
## labor 1400
## steel   50
```

```
factory %*% output[colnames(factory)]
```

```
##       [,1]
## labor 1600
## steel   70
```

```
all(factory %*% output[colnames(factory)] <= available[rownames(factory)])
```

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

- Last lines don't have to change if we add motorcycles as output or rubber and glass as inputs (abstraction again)

Doing the same thing to each row or column

- Take the mean: `rowMeans()`, `colMeans()`: input is matrix, output is vector. Also `rowSums()`, etc.
- `summary()`: vector-style summary of column

```r
colMeans(factory)
```

```
##   cars trucks
##   20.5   31.5
```

```r
summary(factory)
```

```
##       cars           trucks     
##  Min.   : 1.00   Min.   : 3.00  
##  1st Qu.:10.75   1st Qu.:17.25  
##  Median :20.50   Median :31.50  
##  Mean   :20.50   Mean   :31.50  
##  3rd Qu.:30.25   3rd Qu.:45.75  
##  Max.   :40.00   Max.   :60.00  
```

apply()

- Takes 3 arguments: the array or matrix, then 1 for rows and 2 for columns, then name of the function to apply to each

```
rowMeans(factory)
```

```
## labor steel
##    50     2
```

```
apply(factory,1,mean)
```

```
## labor steel
##    50     2
```

- What would `apply(factory,1,sd)` do?
- More on `apply()` later

Lists

- Sequence of values, *not* necessarily all of the same type

```
my.distribution <- list("exponential",7,FALSE)
my.distribution
```

```
## [[1]]
## [1] "exponential"
##
## [[2]]
## [1] 7
##
## [[3]]
## [1] FALSE
```

- Most of what you can do with vectors you can also do with lists

Accessing pieces of lists

- Can use [ ] as with vectors
- Or use [[ ]], but only with a single index
  - [[ ]] drops names and structures, [ ] does not

```
is.character(my.distribution)
```

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

```
is.character(my.distribution[[1]])
```

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

```
my.distribution[[2]]^2
```

```
## [1] 49
```

- What happens if you try `my.distribution[2]^2`?
- What happens if you try [[ ]] on a vector?

Expanding and contracting lists

- Add to lists with `c()` (also works with vectors)

```
my.distribution <- c(my.distribution,7)
my.distribution
```

```
## [[1]]
## [1] "exponential"
##
## [[2]]
## [1] 7
##
## [[3]]
## [1] FALSE
##
## [[4]]
## [1] 7
```

Expanding and contracting lists

- Chop off the end of a list by setting the length to something smaller (also works with vectors)

```
length(my.distribution)
```

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

```
length(my.distribution) <- 3
my.distribution
```

```
## [[1]]
## [1] "exponential"
##
## [[2]]
## [1] 7
##
## [[3]]
## [1] FALSE
```

Naming list elements

- We can name some or all of the elements of a list

```r
names(my.distribution) <- c("family","mean","is.symmetric")
my.distribution
```

```
## $family
## [1] "exponential"
##
## $mean
## [1] 7
##
## $is.symmetric
## [1] FALSE
```

```r
my.distribution[["family"]]
```

```
## [1] "exponential"
```

```r
my.distribution["family"]
```

```
## $family
## [1] "exponential"
```

Naming list elements

- Lists have a special short-cut way of using names, $ (which removes names and structures)

```
my.distribution[["family"]]
```

```
## [1] "exponential"
```

```
my.distribution$family
```

```
## [1] "exponential"
```

Naming list elements

1. Creating a list with names
2. Adding named elements
3. Removing a named list element, by assigning it the value `NULL`

```
another.distribution <- list(family="gaussian",mean=7,sd=1,is.symmetric=TRUE)

my.distribution$was.estimated <- FALSE
my.distribution[["last.updated"]] <- "2011-08-30"

my.distribution$was.estimated <- NULL
```

Key-Value pairs

- Lists give us a way to store and look up data by *name*, rather than by *position*
- Really useful programming concept with many names: **key-value pairs**, **dictionaries**, **associative arrays**, **hashes**
- If all our distributions have components named `family`, we can look that up by name, without caring where it is in the list

Dataframes

- Dataframe = the classic data table, n rows for cases, p columns for variables
- Lots of the really-statistical parts of R presume data frames
  - `penn` from last time was really a dataframe
- Not just a matrix because *columns can have different types*
- Many matrix functions also work for dataframes (`rowSums()`, `summary()`, `apply()`)
  - but no matrix multiplying dataframes, even if all columns are numeric

Dataframes

```
a.matrix <- matrix(c(35,8,10,4),nrow=2)
colnames(a.matrix) <- c("v1","v2")
a.matrix
```

```
##      v1 v2
## [1,] 35 10
## [2,]  8  4
```

```
a.matrix[,"v1"]  # Try a.matrix$v1 and see what happens
```

```
## [1] 35  8
```

Dataframes

```
a.data.frame <- data.frame(a.matrix,logicals=c(TRUE,FALSE))
a.data.frame
```

```
##   v1 v2 logicals
## 1 35 10     TRUE
## 2  8  4    FALSE
```

```
a.data.frame$v1
```

```
## [1] 35  8
```

```
a.data.frame[,"v1"]
```

```
## [1] 35  8
```

```
a.data.frame[1,]
```

```
##   v1 v2 logicals
## 1 35 10     TRUE
```

```
colMeans(a.data.frame)
```

```
##       v1       v2 logicals
##     21.5      7.0      0.5
```

Adding rows and columns

- Can add rows or columns to an array or data-frame with `rbind()` and `cbind()`, but be careful about forced type conversions

```
rbind(a.data.frame,list(v1=-3,v2=-5,logicals=TRUE))
```

```
##   v1 v2 logicals
## 1 35 10     TRUE
## 2  8  4    FALSE
## 3 -3 -5     TRUE
```

```
rbind(a.data.frame,c(3,4,6))
```

```
##   v1 v2 logicals
## 1 35 10        1
## 2  8  4        0
## 3  3  4        6
```

Structures of Structures

- So far, every list element has been a single data value
- List elements can be other data structures, e.g., vectors and matrices

```
plan <- list(factory=factory, available=available, output=output)
plan$output
```

```
## trucks   cars
##     20     10
```

- Internally, a dataframe is basically a list of vectors

Structures of Structures

- List elements can even be other lists
  - which may contain other data structures
    - including other lists . . . which may contain other data structures
- This **recursion** lets us build arbitrarily complicated data structures from the basic ones
- Most complicated objects are (usually) lists of data structures

Example: Eigenstuff

- `eigen()` finds eigenvalues and eigenvectors of a matrix
- Returns a list of a vector (the eigenvalues) and a matrix (the eigenvectors)

```
eigen(factory)
```

```
## eigen() decomposition
## $values
## [1] 41.556171  1.443829
##
## $vectors
##             [,1]       [,2]
## [1,] 0.99966383 -0.8412758
## [2,] 0.02592747  0.5406062
```

```
class(eigen(factory))
```

```
## [1] "eigen"
```

Example: Eigenstuff

- With complicated objects, you can access parts of parts (of parts. . .)

```r
factory %*% eigen(factory)$vectors[,2]
```

```
##             [,1]
## labor -1.2146583
## steel  0.7805429
```

```r
eigen(factory)$values[2] * eigen(factory)$vectors[,2]
```

```
## [1] -1.2146583  0.7805429
```

```r
eigen(factory)$values[2]
```

```
## [1] 1.443829
```

```r
eigen(factory)[[1]][[2]] # NOT [[1,2]]
```

```
## [1] 1.443829
```

Creating an example dataframe

```
library(datasets)
states <- data.frame(state.x77, abb=state.abb, region=state.region, division=state.division)
```

- `data.frame()` is combining here a pre-existing matrix (`state.x77`), a vector of characters (`state.abb`), and two vectors of qualitative categorical variables (**factors**; `state.region`, `state.division`)
- Column names are preserved or guessed if not explicitly set

Creating an example dataframe

```r
colnames(states)
```

```
##  [1] "Population" "Income"     "Illiteracy" "Life.Exp"   "Murder"
##  [6] "HS.Grad"    "Frost"      "Area"       "abb"        "region"
## [11] "division"
```

```r
states[1,]
```

```
##         Population Income Illiteracy Life.Exp Murder HS.Grad Frost  Area abb
## Alabama       3615   3624        2.1    69.05   15.1    41.3    20 50708  AL
##         region           division
## Alabama  South East South Central
```

Dataframe access

- By row and column index

```r
states[49,3]
```

```
## [1] 0.7
```

- By row and column names

```r
states["Wisconsin","Illiteracy"]
```

```
## [1] 0.7
```

Dataframe access

- All of a row:

```
states["Wisconsin",]
```

```
##           Population Income Illiteracy Life.Exp Murder HS.Grad Frost  Area abb
## Wisconsin       4589   4468        0.7    72.48      3    54.5   149 54464  WI
##                  region           division
## Wisconsin North Central East North Central
```

Dataframe access

- All of a column:

```
head(states[,3])
```

```
## [1] 2.1 1.5 1.8 1.9 1.1 0.7
```

```
head(states[,"Illiteracy"])
```

```
## [1] 2.1 1.5 1.8 1.9 1.1 0.7
```

```
head(states$Illiteracy)
```

```
## [1] 2.1 1.5 1.8 1.9 1.1 0.7
```

Dataframe access

- Rows matching a condition:

```
states[states$division=="New England", "Illiteracy"]
```

```
## [1] 1.1 0.7 1.1 0.7 1.3 0.6
```

```
states[states$region=="South", "Illiteracy"]
```

```
##  [1] 2.1 1.9 0.9 1.3 2.0 1.6 2.8 0.9 2.4 1.8 1.1 2.3 1.7
```

Dataframe access

- Parts or all of the dataframe can be assigned to

```
summary(states$HS.Grad)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   37.80   48.05   53.25   53.11   59.15   67.30
```

```
states$HS.Grad <- states$HS.Grad/100
summary(states$HS.Grad)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.3780  0.4805  0.5325  0.5311  0.5915  0.6730
```

```
states$HS.Grad <- 100*states$HS.Grad
```

with()

- What percentage of literate adults graduated HS?

```
head(100*(states$HS.Grad/(100-states$Illiteracy)))
```

```
## [1] 42.18590 67.71574 59.16497 40.67278 63.29626 64.35045
```

- `with()` takes a data frame and evaluates an expression "inside" it

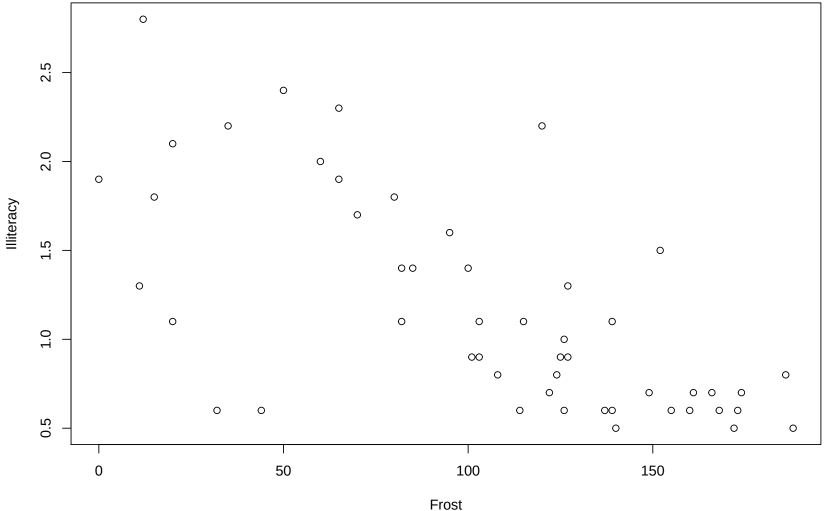
```
with(states, head(100*(HS.Grad/(100-Illiteracy))))
```

```
## [1] 42.18590 67.71574 59.16497 40.67278 63.29626 64.35045
```

Data arguments

- Lots of functions take `data` arguments, and look variables up in that data frame

```
plot(Illiteracy~Frost, data=states)
```



Summary

- Arrays add multi-dimensional structure to vectors
- Matrices act like you'd hope they would
- Lists let us combine different types of data
- Dataframes are hybrids of matrices and lists, for classic tabular data
- Recursion lets us build complicated data structures out of the simpler ones