Lecture 2: More Data Structures

Statistical Computing, 36-350

Wednesday September 2, 2015

Outline

- Arrays
- Matrices
- Lists
- Data frames
- Structures of structures

Vector structures, starting with arrays

Many data structures in R are made by adding bells and whistles to vectors, i.e., they are "vector structures"

Most useful: **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, ... arrays; `dim` is vector of arbitrary length

Some properties of our array:

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

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

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

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

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

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

```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 array 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 indexing arrays

Can access a 2d array either by pairs of indices or by the underlying vector:

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

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

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

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

Omitting an index means "all of it":

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

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

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

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

Functions on arrays

Many functions applied to a vector-structure like an array will just boil things down to the underlying vector:

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

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

This happens unless the function is set up to handle arrays specifically

---

Many functions *do* preserve array structure:

```r
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:

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

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

(We will see a lot more of this idea soon)

Example: houses prices in Pennsylvania

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

```r
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

---

It turns out 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?

We 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, again?)

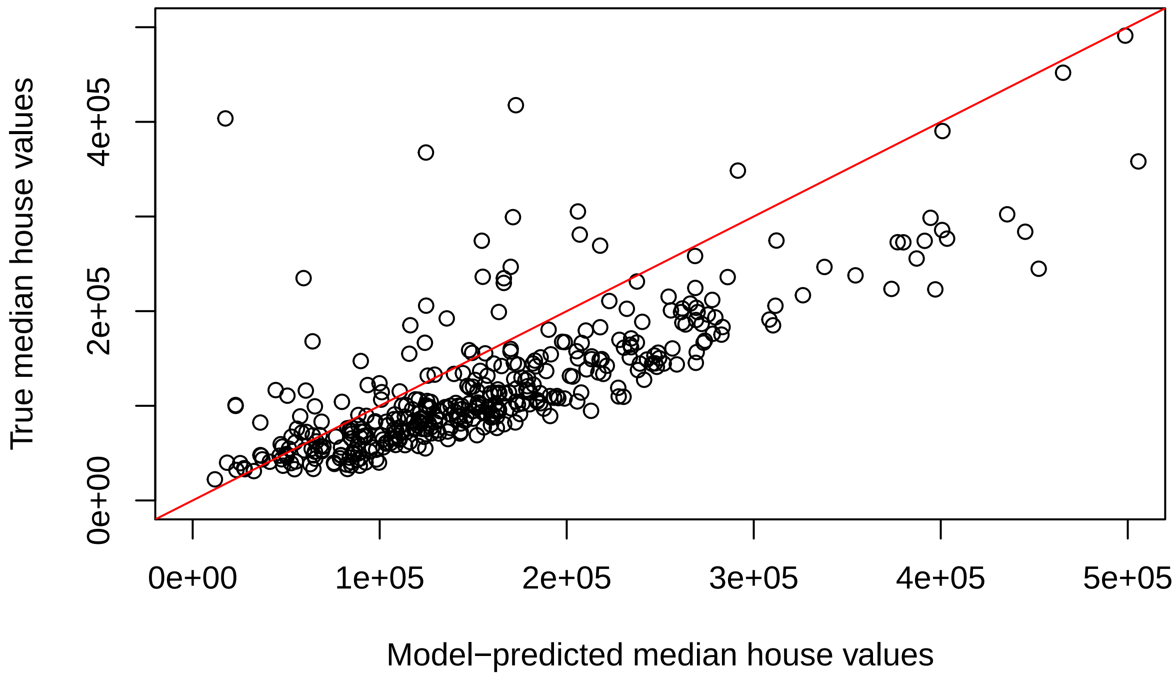
Use variables and names

```
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
```

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



Running example: resource allocation

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

In R, a matrix is a specialization of a 2d array

```
factory = matrix(c(40,1,60,3), nrow=2)
factory
```

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

```
is.array(factory)
```

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

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

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

could also specify `ncol`; to fill by rows, use `byrow=TRUE`

Elementwise operations with the usual arithmetic and comparison operators (e.g., `factory/3`)

Compare whole matrices with `identical()` or `all.equal()`

Matrix multiplication

Has its own special operator, written `%*%`:

```
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:

```
output = c(10,20)
factory %*% output
```

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

```
output %*% factory
```

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

(R silently casts the vector as either a 1-column or 1-row matrix, as appropriate)

Matrix operators

Transpose:

```
t(factory)
```

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

Determinant:

```
det(factory)
```

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

The matrix diagonal

The `diag()` function can be used to extract the diagonal entries of a matrix:

```
diag(factory)
```

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

It can also be used to change the diagonal:

```
diag(factory) = c(35,4)
factory
```

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

Re-set it for later:

```
diag(factory) = c(40,3)
```

Creating a diagonal or identity matrix

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

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

```
diag(2)
```

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

(How do you get a 1 x 1 matrix containing a single entry 2?)

Inverting a matrix

```
solve(factory)
```

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

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

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

Why is it called "solve" anyway?

Solving the linear system $Ax = b$ for x:

```
available = c(1600,70)
solve(factory,available)
```

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

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

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

Names in matrices

We can name either rows or columns or both, with `rownames()` and `colnames()`

These are 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

Names can be used to coordinate different objects

```
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")
```

```r
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
```

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

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

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

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

Note that 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()`, `colSums`

`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 a 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?)

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

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

```
## [[1]]
## [1] 7
```

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

```
## [1] 7
```

```
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
```

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
```

Pluck out all but one piece of a list (also works with vectors):

```
my.distribution[-2]
```

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

(What happens if you try `my.distribution[[-2]]`?)

Naming list elements

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

```
names(my.distribution) = c("family","mean","is.symmetric")
my.distribution
```

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

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

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

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

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

Lists have a special shortcut way of using names, with `$`:

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

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

```
my.distribution$family
```

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

Names in lists (continued)

Creating a list with names:

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

Adding named elements:

```
my.distribution$was.estimated = FALSE
my.distribution[["last.updated"]] = "2015-09-01"
```

Removing a named list element, by assigning it the value `NULL`:

```
my.distribution$was.estimated = NULL
```

Key-value pairs

Lists give us a natural way to store and look up data by *name*, rather than by *position*

A really useful programming concept with many names: **key-value pairs**, **dictionaries**, **associative arrays**

If all our distributions have components named `family`, we can look that up by name, without caring where it is (in what position it lies) in the list

Data frames

The classic data table, n rows for cases, p columns for variables

Lots of the really-statistical parts of R presume data frames

Not just a matrix because *columns can have different types*

Many matrix functions also work for data frames (e.g.,`rowSums()`, `summary()`, `apply()`)

(But no matrix multiplication with data frames, even if all columns are numeric!)

```
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
```

```
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

We 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 data frame is basically a list of vectors (all of the same length)

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: eigen-decomposition

`eigen()` finds eigenvalues and eigenvectors of a matrix

Returns a list of a vector (the eigenvalues) and a matrix (the eigenvectors)

```
eigen(factory)
```

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

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

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

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
```

Summary

- Arrays add multi-dimensional structure to vectors
- Matrices act like you'd hope they would
- Lists let us combine different types of data
- Data frames are hybrids of matrices and lists, allowing each column to have a different basic type
- Recursion lets us build complicated data structures out of simpler ones