Section 2: R and matrix madness

Ed Rubin

Contents

1 Admin

1.1 What you will need

1. The packages and data files from the previous lecture. The packages are `readr`, `dplyr`, and `haven`. (Run `install.packages(c("readr", "dplyr", "haven"))`)
2. The package `psych`. (Run `install.packages("psych")`.)

1.2 Summary of last time

- Installing and loading packages
- File paths (finding, changing, defining)
- Loading data (`.dta` and `.csv`)

Follow up:

- My office hours are Mondays 3:30pm–5:30pm, Giannini Hall, room 236.
- Review last section's notes on indexing.

- To create a folder, use the `dir.create()` function. *E.g.* `dir.create("TestFolder")` should create a new folder named "TestFolder" in your current directory. To see if it worked, type `dir()`. You can also check whether a folder already exists using the `dir.exists()` function, *e.g.* `dir.exists("TestFolder")`.
- A shortcut to clear your RStudio console: `ctrl+L`.
- If you have any questions about LaTeX and knitr, please check out my summary on using RStudio, LaTeX, and knitr. I'm happy to field questions. Google works too.

1.3 Summary of this section

Data structures in R—vectors and matrices in detail.

2 Data structures in R

As we discussed previously, when you load or create data in R, you create objects. As you could probably guess, there are different types of objects in R. For the most part, we will use four types of objects in this course. Today we will focus on vectors and matrices today.

1. **vectors**
2. **matrices**
3. data frames (or similar objects, like last lecture's `tibble`)
4. lists

Each of these object types is a different way to store data. Each can take numbers or characters. And you can generally change the type of an object fairly easily. Plus, they all follow similar indexing rules,[1] which is quite nice.

While these object types have much in common, they each act slightly differently, so it is important to keep track of your object types. Luckily, R provides us with the `class()` function, as well as `dim()` and `length()`, which both help us figure out the type of data object we have. You can also check the "Environment" tab in RStudio to see the objects that R currently holds in memory and their types.

First, check the class of a number

```
class(2)
```

```
## [1] "numeric"
```

Now, let's check the class of a string of characters

```
class("Max Auffhammer")
```

```
## [1] "character"
```

[1] Check out Section 1 for more on indexing.

3 Vectors

3.1 Numeric vectors

Vectors are more-or-less the basic building block in R. Vectors are one dimensional, and they only need one element. One way to create a vector is using the `c()` (*combine*) function.

```
# Create a vector called 'vec'
vec <- c(1, 2, 3, 4, 5)
# Print the vector named 'vec'
vec
```

```
## [1] 1 2 3 4 5
```

To create (consecutive) sequences of (natural) numbers, you can use the colon (`:`), *e.g.* `1:5`. You can also use the `seq()` function, *e.g.*, `seq(from = 1, to = 5)` or `seq(from = 1, to = 5, by = 1)`, which gives you flexibility in the increments between numbers.

```
# Create a vector of the sequence from 1 to 5 and store it as 'vec2'
vec2 <- 1:5
# Print 'vec2'
vec2
```

```
## [1] 1 2 3 4 5
```

```
# Are the two vectors' elements equal?
vec2 == vec
```

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

```
# Are the two vectors equal?
all.equal(vec, vec2)
```

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

Notice that using the double-equal sign (==) tests pairwise equality, while the function `all.equal()` tests equality between the two vectors.[2]

```
# Create another numeric vector
vec3 <- c(1, 2, 8:10)
# Print it
vec3
```

```
## [1]  1  2  8  9 10
```

```
# Check element-wise equality
vec == vec3
```

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

```
# Check overall equality
all.equal(vec, vec3)
```

[2] You can also use the `identical()` function, which is probably more appropriate, but beware: it is *very* picky. For instance, try `as.integer(1) == as.numeric(1)` and `all.equal(as.integer(1), as.numeric(1))` compare the results to `identical(as.integer(1), as.numeric(1))`. Picky! Might not always get you what you want. User beware.

```
## [1] "Mean relative difference: 1.25"
```

3.2 Combining vectors

You can also create vectors out of vectors. The combine function `c()` simply concatenates the vectors.

```
# Create a new vector by combining 'vec2' and 'vec3'
vec23 <- c(vec2, vec3)
# Print the new vector
vec23
```

```
##  [1]  1  2  3  4  5  1  2  8  9 10
```

Recall that vectors are one-dimensional. Let's explore our vector a bit using the functions `class()`, `is.vector()`, `dim()`, and `length()`.[3]

```
# The class function
class(vec23)
```

```
## [1] "numeric"
```

```
# The is.vector function
is.vector(vec23)
```

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

```
# The dimension function
dim(vec23)
```

```
## NULL
```

```
# The length function
length(vec23)
```

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

Because vectors are one-dimensional, indexing the vectors requires only one input. Let's grab the seventh element of the vector `vec23`

```
vec23[7]
```

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

3.3 Character vectors

Finally, we can also create vectors of characters.[4]

```
# Create the string vector using quotation marks
str_vec <- c("Aren't", "vectors", "exciting", "?")
# Print it
str_vec
```

[3] Most object types/classes have an associated function like the `is.vector()` function. For example, `is.numeric()`, `is.integer()`, `is.character()`, `is.matrix()`, and `is.data.frame()`, to name a few.

[4] Recall that we use quotation marks to create string/character vectors.

```
## [1] "Aren't"   "vectors"  "exciting" "?"
```

```
# Check its class
class(str_vec)
```

```
## [1] "character"
```

```
# Check if it is a vector
is.vector(str_vec)
```

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

```
# Grab the third element
str_vec[3]
```

```
## [1] "exciting"
```

What happens when you combine a vector of characters with a vector of numbers (or create a vector of both numbers and characters)?

```
# Create a vector of the numeric vector 'vec' and the character vector 'str_vec'
mix_vec <- c(vec, str_vec)
# Print the result
mix_vec
```

```
## [1] "1"        "2"        "3"        "4"        "5"        "Aren't"
## [7] "vectors"  "exciting" "?"
```

```
# Check the class of the new vector
class(mix_vec)
```

```
## [1] "character"
```

4 Matrices

It's matrix time.

4.1 `matrix()`

R's matrix function is aptly named `matrix()`. We will generally feed the matrix function two arguments:[5]

1. `data`: the *stuff* inside the matrix
2. `ncol`: the number of columns[6]

To learn more about the matrix function, type `?matrix` in your console (or in the "Help" tab of RStudio). This help searching works for all loaded functions. If you do not find what you are looking for, try using two question marks, `??matrix`, which will perform a fuzzy search for the word.

[5] The `matrix()` function, in fact, requires only one argument: `data`. Without specifying `ncol` or `nrow`, `matrix()` returns a $n \times 1$ matrix.

[6] Alternatively, you can use `nrow` and omit `ncol`. Or use both… or neither.

4.2 Creating matrices

Let's make a matrix. Specifically, let's make a 3×2 matrix filled with the numbers between 1 and 6. Following Max's notation, let's call this matrix $\mathbf{A}$.

```
# Create a 3x2 matrix filled with the sequence 1:6
A <- matrix(data = 1:6, ncol = 2)
# Print it
A
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

What are the dimensions of `A`? What about its length? Is `A` a vector or a matrix?

```
# The dimension of A
dim(A)
```

```
## [1] 3 2
```

```
# The length of A
length(A)
```

```
## [1] 6
```

```
# Check if A is a vector
is.vector(A)
```

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

```
# Check if A is a matrix
is.matrix(A)
```

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

Finally, you can index any element of a matrix using its row *and* column. The way R prints matrices actually shows you exactly how to reference the row, the column, or the exact element.

```
# Print the matrix A
A
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
# Print the second row of A
A[2,]
```

```
## [1] 2 5
```

```
# Print the second and third rows of A
A[2:3,]
```

```
##      [,1] [,2]
## [1,]    2    5
## [2,]    3    6
```

```
# Print the first column of A
A[,1]
```

```
## [1] 1 2 3
```

```
# Print the element in the second row and first column of A
A[2,1]
```

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

4.3 Transpose

Define $\mathbf{B} = \mathbf{A}'$. How can we manually make the transpose of $\mathbf{A}$? It should be 2×3. Let's try

```
# Print A
A
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
# First attempt at B
B <- matrix(data = 1:6, ncol = 3)
# Print B
B
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Nope. The dimensions are correct, but we want the first row to be 1, 2, 3—not 1, 3, 5. To tell R to fill the matrix by row, rather than by column (the default), use the argument `byrow`.

```
# Using the byrow option
B <- matrix(data = 1:6, ncol = 3, byrow = TRUE)
# Print B
B
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

That's better.

R has a much simpler way to transpose matrices: the `t()` function. Feed a matrix to the `t()` function, and the output will be the transpose of the matrix. Let's verify.

```
# Check element by element
t(A) == B
```

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

```
# Check if all the element-by-element comparisons are TRUE
all(t(A) == B)
```

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

```
# Check if the transpose of A is identical to B
identical(t(A), B)
```

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

4.4 Multiplication

In R, the matrix multiplication command is `%*%`. Admittedly, this notation is a little strange,[7] but `*` is already taken for scalar multiplication. Don't get lazy here: R will still let you use `*` multiplication with matrices, but it will be **elementwise** multiplication—not matrix multiplication.

As an example, here is the elementwise multiplication of $\mathbf{A}$ and itself, using the `*` operator

```
# Print A as a reminder
A
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
# Elementwise multiplication of A and A
A * A
```

```
##      [,1] [,2]
## [1,]    1   16
## [2,]    4   25
## [3,]    9   36
```

The matrix multiplication of $\mathbf{A}$ and itself is not defined because $\mathbf{A}$ does not conform with itself.

```
# Matrix multiplication of A and itself
A %*% A
```

```
## Error in A %*% A: non-conformable arguments
```

You can also multiply matrices by scalars using the `*` operator

```
A * 3
```

```
##      [,1] [,2]
## [1,]    3   12
## [2,]    6   15
## [3,]    9   18
```

[7] In case you are wondering: I did not choose this notation.

To see a matrix multiplication that actually works, let's define a 2×2 matrix with the elements 1 through 4. Call it $\mathbf{C}$. Calculate $\mathbf{A} \times \mathbf{C}$.

```
# Create C
C <- matrix(data = 1:4, ncol = 2)
# Multiply A and C
A %*% C
```

```
##      [,1] [,2]
## [1,]    9   19
## [2,]   12   26
## [3,]   15   33
```

4.5 Identities and inverses

No discussion of matrices would be complete without identity matrices and inverses.

In R, we create identity matrices using the function `diag()`. The argument to `diag()` is the number of rows/columns of the identity matrix. Because identity matrices are square, this argument is a single number.

For a 5×5 identity matrix,

```
diag(5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

Notice that if you apply the function `diag()` to an already-created matrix, it will grab the diagonal elements of that matrix.

```
# Grab the diagonal elements of the matrix C
diag(C)
```

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

R's built-in function to take the inverse of a matrix is `solve()`. Give `solve()` an invertible matrix, and it will return the inverse of the matrix.

```
# Take the inverse of C
solve(C)
```

```
##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5
```

```
# Multiply C by its inverse
C %*% solve(C)
```

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

The function `det()` will give you the determinant of a (square) matrix. To calculate the trace of a matrix, you need to load the `psych` package. Within the `psych` package is a function called `tr()` that will calculate the trace.

```
# The determinant of a matrix
det(C)
```

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

```
# The trace of a matrix using the psych package's tr()
library(psych)
tr(C)
```

```
## [1] 5
```

What is another way to calculate the trace of a matrix using tools we've already learned?

```
sum(diag(C))
```

```
## [1] 5
```

Combining `sum()` and `diag()` will give the trace of a matrix.

4.6 Adding columns or rows

You will often want to add an additional row or column to your matrix. `rbind()` and `cbind()` bind rows and columns onto existing matrices, respectively.

Remember the matrix $\mathbf{A}$?

```
A
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Let's add a column of ones in front of the matrix.

```
# Bind a column of ones in front of the matrix A
cbind(c(1, 1, 1), A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    4
## [2,]    1    2    5
## [3,]    1    3    6
```

What happens if we get a little lazy and type `1` instead of `c(1, 1, 1)`?

```
cbind(1, A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    4
## [2,]    1    2    5
## [3,]    1    3    6
```

R recycles the data given to make the dimensions match. This feature can be very helpful, but it might occasionally trip you up too. Be careful being lazy.

Now, let's bind the vector `c(3, 6)` to the end (bottom?) of A.

```
rbind(A, c(3, 6))
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
## [4,]    3    6
```

Finally, notice that you can bind matrices together (as long as the dimension match).

```
rbind(A, C)
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
## [4,]    1    3
## [5,]    2    4
```

5 Other tools

We do not have time to cover every function related to vectors and matrices, but here are a few more functions (or uses of functions we covered) that may prove useful.

- `crossprod()` and `tcrossprod()` take cross products (*e.g.*, $\mathbf{A}' \times \mathbf{B}$).
- `solve()` can also take two arguments, `a` and `b`, and returns the solution for `x` in the equation `a %*% x = b`.
- `eigen()` gives the eigenvectors and eigenvalues of a matrix. `eigen(A)$vectors` will give you just the eigenvectors of $\mathbf{A}$, and `eigen(A)$values` will give you only the eigenvalues of $\mathbf{A}$.
- You can give `matrix()` a single number and specify both `ncol` and `nrow` to fill a matrix with the same number. For example, `matrix(data = 1, nrow = 3, ncol = 5)` creates a 3×5 matrix of ones.
- The function `sample()` will randomly sample a number of elements from a vector with or without replacement (without replacement is the default). For example, `sample(x = 1:100, size = 5, replace = TRUE)` will randomly draw five numbers from 1 to 100 with replacement. See `?sample` for more information. You should set your seed to make sampling replicable. The function `set.seed()` sets the seed.