Matrix Basics

Sam Mason

Creating Matrices

Matrices are commonly built using the `matrix()` function.

Task 1

Create a square matrix containing the first 100 integers (counting numbers from 1 to 100), filled by row in reverse order. That is, if your matrix is called `task1` then `task[1, 1]` should hold the value `100`, `task1[1, 2]` should the value `99`, etc.

We can also squish a bunch of vectors together to form matrices using the `cbind()` and `rbind()` function.

Task 2

We've used the `rnorm()` function several times to generate random data drawn from a normal distribution. Create the following vectors of random data drawn from various other probability distributions. We'll talk about all of these distributions in much greater detail in the weeks to come.

1. 50 draws from a uniform distribution ranging from 1 to 10 (`runif()`)
2. 50 draws from a Poisson distribution with `lambda = 5` (`rpois()`)
3. 50 draws from a normal distribution with `mean = 5` and `sd = 1`
4. 50 draws from a binomial distribution with `size = 10` and `prob = 0.5` (`rbinom()`)

Task 3

Combine all of these vectors into a new matrix using the `cbind()` function. Set the column names to the following character vector: "'c("Uniform", "Poisson", "Normal", "Binomial").

Task 4

Let's generate some new observations (rows), and tack them on to end of our matrix. Rerun all the code used to generate the vectors in **Task 2**. This should produce four new random vectors. Using both `cbind()` and `rbind()`, add this new data to the existing data to produce a 100 row by 4 column matrix where the last 50 rows are filled with the new values generated in this task. To confirm the dimension of the new matrix, call the `dim()` function, which returns the number of rows and columns in a given matrix.

Challenge Task 1

In a single line of code create a matrix where the 50 rows of new data generated in **Task 4** are sandwiched between the first 25 and last 25 rows of the matrix produced in **Task 3**.

Matrix Operations

R is jam-packed with functions, but some shine brighter than the rest. The `apply()` function is one of those. A real superstar in the R world. This function allows us to *apply* almost any function we'd like to the margins of a matrix. We can, for example, replicate the `rowSums()` function you used in DataCamp using the `apply()`.

```
mat <- matrix(1:9, nrow = 3, ncol = 3)
rowSums(mat)
```

```
## [1] 12 15 18
```

```
apply(mat, MARGIN = 1, FUN = sum) # MARGIN = 1 indicates that the sum function
```

```
## [1] 12 15 18
```

```
                                  # is to be applied to each row
```

For now we'll only be using functions included in base R, but the true power of `apply()` will be unlocked once we learn how to write our own function from scratch!

Task 5

Using the `apply()` function, calculate the column means for the matrix you produced in **Task 4**. The mean for all of these distributions (the population mean) is 5. The means that you've calculated (sample means) are probably not exactly 5. Which columns deviate the most from the population mean?

Task 6

Calculate the standard deviation of each column. How does this output help to explain why certain distributions produced more deviant samples?

Indexing & Subsetting Matrices

Like vectors, we index matrices using square brackets. Matrices, however, have two dimensions to index, meaning we need to indicate the row(s) and column(s) we'd like to select from the matrix.

```
mat[1, 3] # matrices are indexed by the row first, and then the column
```

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

```
mat[, 3] # leaving the row index blank means "select all rows"
```

```
## [1] 7 8 9
```

```
mat[1, ] # leaving the column index blank means "select all columns"
```

```
## [1] 1 4 7
```

```
mat[1, 1:3] # we could equivalently explicitly select all columns
```

```
## [1] 1 4 7
```

Task 7

Create a 5 by 5 matrix containing the first 25 integers. Print out only the odd-numbered rows (1, 3, 5).

Task 8

This task is simple enough with a small matrix, but becomes a little more tedious with a very large matrix. Produce a 100 by 5 matrix containing the first 500 integers. I still want you print out every odd-numbered row, but this time use the `seq()` function to make your life a little easier.

Task 9

Just like we filtered vectors using Boolean operators, we can filter matrices. Using the matrix produced in **Task 4**, create a Boolean vector that is `TRUE` when the "Poisson" column is less than the mean of that column plus one standard deviation.

Task 10

Using that logical vector, index the **Task 4** matrix to yield a matrix containing only those rows that include "Poisson" values less than the mean of that column plus one standard deviation. You can check to see if you've indexed properly by asking R the same true-or-false question you used to generate the logical vector in **Task 9**, but this time using the newly indexed matrix. All elements of the resulting logical vector should be `TRUE`. You can check this by calling `sum()` on the new logical vector. In R, `TRUE` = 1 and `FALSE` = 0.

Task 11

How could accomplish this same kind of Boolean filtering using the `subset()` function?

Task 12

To make sure the two subsetted matrices from **Task 10** and **Task 11** are identical, subtract one from the other. The resulting matrix should contain only zeros.