STM4PSD – Workshop 4

Calculating probabilities using R

R has a large number of built-in probability distributions. Type `?distributions` in the console to view the list. Note that some of them are continuous distributions, which we will start looking at next week.

For most of the built-in distributions, R defines four functions, which take on the following forms:

- `dxxxx`, the ***D****ensity function* for the distribution. For discrete random variables, this is the same as the probability mass function. We will be seeing density functions in more detail when we look at continuous random variables.
- `pxxxx`, the *cumulative* ***P****robability function* for the distribution, which is used to calculate probabilities of the form $P(X \leqslant a)$, for some given value a.
- `qxxxx`, the ***Q****uantile* function for the distribution. We will discuss this in more detail when we look at continuous distributions, and then later see their applications for confidence intervals and hypothesis testing.
- `rxxxx`, the ***R****andom generation function* for the distribution. This can be used to generate random numbers following a given distribution, which is useful for simulation purposes.

In particular, for the binomial distribution, the characters `binom` take place of `xxxx` above, yielding the `dbinom`, `pbinom`, `qbinom` and `rbinom` functions. For example, suppose $X \sim \text{Bin}(100, 0.1)$.

- You can calculate $P(X = 50)$ by typing `dbinom(50, size=100, prob=0.1)` or just `dbinom(50, 100, 0.1)`.
- You can calculate $P(X \leqslant 50)$ by typing `pbinom(50, size=100, prob=0.1)` or just `pbinom(50, 100, 0.1)`.

Probabilities of the form $P(X > a)$ can be calculated by passing the `lower.tail=FALSE` argument to the `pxxxx` function. For example,

- You can calculate $P(X > 50)$ by typing `pbinom(50, size=100, prob=0.1, lower.tail=FALSE)` or just `pbinom(50, 100, 0.1, FALSE)`.

For probabilities using other forms of inequality, e.g., $P(X \geqslant a)$ or $P(X < a)$, you will need to use properties of probability like those described by Result 2.5.3.

1. Let $X \sim \text{Bin}(100, 0.3)$. Using R, calculate:
   (a) $P(X = 10)$ (b) $P(X \leqslant 10)$ (c) $P(X < 10)$
   (d) $P(X > 10)$ (e) $P(X \geqslant 10)$ (f) $P(5 \leqslant X < 10)$
2. Use R to verify the answers to Question 8 from Workshop 1.
3. Read the documentation for the probability distributions implemented in R by typing `?distributions` in the console, and locate the appropriate function needed for the negative binomial distribution. Then, use R to verify the answers to Question 9 from Workshop 1.
4. Similarly, use R to verify the answers to the problems involving probability from Question 7 of Workshop 1.
5. (a) Write an R function `binom.less`, which takes three arguments: `q`, `size` and `prob`, and calculates $P(X < \texttt{q})$, where $X \sim \text{Bin}(\texttt{size}, \texttt{prob})$. You will need to make use of the `dbinom` function in your code.
   Make use of algebraic properties of probability as described in Result 3.5.3.
   (b) Write an R function `binom.geq`, which takes three arguments: `q`, `size` and `prob`, and calculates $P(X \geqslant \texttt{q})$, where $X \sim \text{Bin}(\texttt{size}, \texttt{prob})$. Here, "geq" is short for "greater than or equal to". You should use the `binom.less` function from (a) in your answer.
   (c) Write an R function `binom.between`, which takes four arguments: `a`, `b`, `size` and `prob`, and then calculates $P(\texttt{a} \leqslant X \leqslant \texttt{b})$, where $X \sim \text{Bin}(\texttt{size}, \texttt{prob})$.

Plotting graphs in R

To plot functions using R, we make use of the `curve` function. A simple example, plotting the function $\cos(x)$ from -4π to 4π is shown below:

```
curve(cos, from = -4*pi, to = 4*pi)
```

Type `?curve` to read the documentation. You'll see that there are various arguments. Here are some useful ones to consider:

- The `from` and `to` arguments indicate the range of values to plot the function over.
- The `xlim` argument specifies the range of values to show on the x-axis, and must be input as a vector of two elements; e.g. `xlim = c(-15,15)` will result in an x-axis ranging from -15 to 15.
- Similarly, the `ylim` argument does the same but for the y-axis.
- The `xlab` and `ylab` arguments allow you to define the labels shown on the x-axis and y-axis.
- The `lwd` argument specifies the *line width*, as a number.
- The `col` argument allows you to add a color for the plot.

Type the following in the console to observe the effects of these arguments:

```
curve(cos, from=-15,
           to=5,
           xlim=c(-20,15),
           xlab="INSERT X LABEL HERE",
           ylab="INSERT Y LABEL HERE",
           lwd=5,
           col="blue")
```

You may also notice that the plot appears somewhat "jagged", especially around the extreme values of the graph. This is because the plotting method for R will sample 101 points by default, and then connect them with straight lines. You can increase the number of points using the `n` argument.

6. Set the value of `n` in the example above to 10000 and observe the difference.

There are several different approaches to plotting functions in R. Two possible ways are:

- Put the formula directly into the `curve` function:

```
curve(x^2+1, from = -10, to = 3)
```

- Define the function separately, and then pass it as an argument to the `curve` function:

```
f <- function(x) { x^2 + 1 }
curve(f, from = -10, to = 3)
```

Generally speaking, the `curve` function requires one of two options as the first argument:

- the name of a function, in which case the first argument of that function will be used for the x-axis, or
- an expression involving the symbol `x`, in which case that variable is used for the x-axis.

The second method is generally more preferable, especially when plotting more complicated functions.

Another approach makes use of anonymous functions (often known in other languages as lambda functions). For example,

```
curve(function(x) { x^2 + 1 }, from = -10, to = 3)
```

You can also overlay more than one graph by including `add = TRUE` in the function arguments:

```
curve(cos, from=-10, to=10, col="blue")
curve(sin, from=-10, to=10, col="red", add=TRUE)
```

7. Use R to create a plot of the function $g(x) = \frac{3}{4}(x+1)(1-x)$, with x values ranging from -3 to 3. Remember that multiplication must be specified using `*`. After creating the plot, you can type `abline(h=0)` to add a horizontal line through $y = 0$, and type `abline(v=0)` to add a vertical line through $x = 0$.

Conditional statements in R

Throughout this week, we have been making use of functions defined using branching statements. For instance, in Example 4.4.2 of the reading notes, we defined the function $f(x)$ given by

$$f(x) = \begin{cases} \frac{(x-1)^2}{3} & \text{if } -1 \leqslant x \leqslant 2, \\ 0 & \text{otherwise.} \end{cases}$$

To code conditional functions, we must make use of `if` statements. The basic syntax of an if statement is the following:

```
if (test-condition) {
    thing to do if condition is true
}
else {
    thing to do if condition is false
}
```

The `test-condition` component must be some kind of condition which evaluates in R to either `TRUE` or `FALSE`. Note that `TRUE` and `FALSE` are (case-sensitive) keywords in R. Various tests can be made, for example:

- You can test for equality by using two equals signs, written `==`. For example `x == 3` will return `TRUE` if the value stored in `x` is equal to 3, and `FALSE` otherwise.
- You can test for non-equality by using `!=`. For example `x != 3` will return `TRUE` if the value stored in `x` is not equal to 3, and `FALSE` otherwise.
- Inequalities such as `<`, `<=`, `>`, `>=` can also be used.

Typing `?Control` and `?Comparison` (case-sensitive) in the R console will bring up the documentation for `if` statements and the relational comparisons.

Actually, the behaviour described above is not *entirely* true. If `x` is a vector, then the comparisons above will also result in a vector, where each element of the resulting vector is the result of performing the comparison elementwise on the vector(s).

8. What do you expect to see when typing `1:6 == 3` in the R console? What about `1:6 >= 3`? Try it for yourself.

9. Try to predict what you will see when you write the following: `0:6 == (0:6)^2`. After evaluating it in R, explain the result.

You can also combine test conditions using "and", "or" and "not":

- `test1 & test2` tests for test1 AND test2, returning `TRUE` only when both `test1` and `test2` evaluate to `TRUE`.
- `test1 | test2` tests for test1 OR test2, returning `TRUE` when at least one of `test1` or `test2` evaluate to `TRUE`.
- `!test1` tests for the negation of test1, returning `TRUE` only when `test1` is false.

You can read more about these statements by typing `?base::Logic` in the R console.

Using this, we could define the function in Example 4.4.2 as follows:

```
f <- function(x) {
    if (x >= -1 & x <= 2) { (x-1)^2/3 }
    else { 0 }
}
```

Consider the logic of the function:

- The inequality $-1 \leqslant x \leqslant 2$ is really an AND statement: $x \geqslant 1$ AND $x \leqslant 2$.
- If this condition is satisfied, we make use of the formula.
- Otherwise, we use the value zero.

10. Load the function above and use it to evaluate $f(-2)$, $f(-1)$, $f(0)$, $f(1)$ and $f(2)$.

Vectorized functions

Recall that many functions in R are vectorized, which means you can do computations like the following:

```
x <- -10:10
(x-1)^2/3
log(x)
```

Using vectors when evaluating expressions like the above will apply the function to each element of the vector.

11. Evaluate both `f(1:6)` and `f(-2:2)` in the console and observe the output. Something will go wrong this time.

The reason this problem occurs is because the plain `if` statement is *not* vectorized; i.e., it does not plan ahead for when vectors may be used as input. A vectorized approach is shown below, using the `ifelse` function.

```
f <- function(x) { ifelse(x >= -1 & x <= 2, (x-1)^2/3, 0) }
```

Note that `ifelse` takes three arguments: the first is a test condition, the second is the expression to use if true, and the third is the expression to use if false. It behaves in exactly the same way as an `if` statement, except it is vectorized.

Vectorization is important for plotting functions, and as we will see soon, for performing integration.

```
# The following will NOT work:
f <- function(x) {
    if (x >= -1 & x <= 2) { (x-1)^2/3 }
    else { 0 }
}
curve(f, from=-10,to=10)

# But this WILL work:
f <- function(x) {
    ifelse(x >= -1 & x <= 2, (x-1)^2/3, 0)
}
curve(f, from=-10,to=10)
```

You will see that the discontinuities are not plotted so well. To (optionally) remedy this, you would need something more intricate, like so:

```
curve(f, from=-1,to=2, xlim=c(-10,10))
lines(x=c(-10, -1), y=c(0,0))
lines(x=c(2, 10), y=c(0,0))
points(x = c(-1,2), y= f(c(-1,2)), pch=19)
points(x = c(-1,2), y = c(0,0), pch=21, bg="white")
```

The reason a vectorized function is required when using `curve` is because, internally, R will first evaluate the function using a large vector of x-values. The size of this vector is what the optional argument `n` adjusts, described earlier. Setting `n` results in a larger vector to evaluate on, which increases the accuracy and smoothness of the graph.

12. Let $f(x)$ be the function defined as follows:

$$f(x) = \begin{cases} 4x - 4 & \text{if } 1 \leqslant x \leqslant 1.5, \\ 8 - 4x & \text{if } 1.5 < x \leqslant 2, \\ 0 & \text{otherwise.} \end{cases}$$

   (a) Define an R function `f` which implements this function. Ensure that the function is vectorized appropriately. Note that `ifelse` statements can be nested.

   (b) Plot the function $f(x)$ in R, over the interval $[0, 2]$.