R code for Actuarial Science courses

Toby Kenney

October 2, 2018

This document describes a lot of R functions and tricks that are useful for the ACSC/STAT 4703 and ACSC/STAT 4720 courses that I teach.

Distribution Functions

`pgamma(x,shape,rate,scale,lower.tail)`

This computes the distribution function of a gamma distribution. `x` is the value at which we want to calculate the distribution function, `shape` is the shape parameter α, `scale` is the scale parameter θ. `lower.tail` is set to `TRUE` (the default) for calculating the distribution function, while setting it to `FALSE` calculates the survival function. Note: calculating the survival function as `1-pgamma(x,shape=alpha,scale=theta)` can be numerically unstable if `x` is large. The rate parameter should not be set — it is used for an alternative parametrisation from the one taught in these courses.

Example: Compute $P(X > 1000)$ for $X \sim$ `Gamma`$(3, 100)$.

```
pgamma(1000,shape=3,scale=100,lower.tail=FALSE)
```

`pnorm(x,mean,sd,lower.tail)`

This computes the distribution function of a normal distribution — the *Phi* function. The default values for `mean` and `sd` are 0 and 1 respectively, so without setting those values, we compute the Φ function. `x` is the value at which we want to calculate the distribution function, `mean` is the mean μ of the normal distribution, `sd` is the standard deviation (**not the variance**) σ. `lower.tail` is set to `TRUE` (the default) for calculating the distribution function, while setting it to `FALSE` calculates the survival function. Note: calculating the survival function as `1-pnorm(x)` can be numerically unstable if `x` is large.

Example: Compute $\Phi(23.4) - \Phi(22.85)$

```
pnorm(22.85,lower.tail=FALSE)-pnorm(23.4,lower.tail=FALSE)
```

Note that we have calculated the survival function (and therefore reversed the order of the values). This is because for these large values, the values of the distribution function will be rounded to 1 to the limit of the machine accuracy, so the difference would be zero.

Vector Operations

Arithmetic operations in `R` and many functions by default operate pointwise on vectors. For example

```
c(2,4,8)^c(4,3,2)
```

will perform the exponentiation elementwise producing the vector

```
c(2^4,4^3,8^2)
```

This is faster than using a loop. You can generate a vector of consecutive values using `start:end`. If you want to produce a vector of `n` consecutive values starting at 1, you should instead use `seq_len(n)`, because this will correctly produce an empty vector if `n=0`, whereas `1:n` will produce a backwards vector `c(1,0)`.

As an example, we can compute the first `l` terms of a convolution of vectors `f` and `g` of length `l` (for the last `l-1` terms, we need to use a different formula

```
for (i in seq_len(l)) {
  convolution[i]<- sum(f[1:i] g[i:1])
}
```

This is much faster than the loop

```
for (i in seq_len(l)) {
  convolution[i]<-0
  for(j in seq_len(i)){
    convolution[i]<- convolution[i]+(f[j] g[i+1-j])
  }
}
```

Example: Computing empirical distribution functions:

```
empirical_distribution<-function(x,data){
  ### data is the data from which to calculate the function
  ### x is the value at which to compute the distribution function
  ### This works pointwise if x is a vector
  comparisons<-rep(1,length(x))%*%t(data)<=x%*%rep(1,length(data))
  nless<-rowSums(comparisons)
  return(nless/length(data))
}
```

This function works by constructing a matrix of comparisons. The `%*%` operator is matrix multiplication. When we multiply a column vector by a row vector, the result is a matrix. In this case `rep(1,length(x))%*%t(data)` produces a matrix whose `[i,j]` entry is `data[j]` and `x%*%rep(1,length(data))` produces a matrix whose `[i,j]` entry is `x[i]`. The matrix `comparisons` therefore produces a logical matrix whose `[i,j]` entry is `data[j]<=x[i]`. By taking the rowSums of this matrix, we count the number of data points smaller than the entries of x.