STATISTICAL MODELLING Part IIC, Michaelmas 2024
Practical 2: Linear Models

For Loops and While Loops

Often we will like to run the same code many times, for instance if we using simulated data to examine a statistical methodology. The prime tools to do this are *for* and *while* loops in the code. These loops repeat the code inside the { } blocks then iterates until the for loop runs out of indices or the while loop condition is broken. These have similar syntax:

```
for (i in 1:B) { doSomething() }

counter <- 0
while (counter < B) {
   doSomething()
   counter <- counter + 1
}
```

both of which accomplish the same goal of calling the `doSomething` function B times.

This approach applies the code blocks in sequence, which is useful if accessing the same memory block and adjusting it multiple times (which is what happened to the memory storing i in the while loop). However often the code will access new memory for each iteration, and so we could instead run these in parallel. For further reading on this, search for the documentation on the `parallel` package in R.

Vectorisation of Code

Consider the computing the column means of the elements in a large matrix X using a for loop:

```
n <- 100
d <- 3000

X <- matrix( rnorm(n * d), nrow = n, ncol = d )
X_means <- rep(0, d)
for (i in 1:d) {
  X_means[i] <- mean(X[,i])
}
```

we can use the `system.time` function to test how long this computation takes:

```
system.time( for (i in X) { X_means[i] <- mean(X[,i]) } )
   user  system elapsed
 43.518   7.317  51.124
```

In contrast, we can compute this much more quickly by using the `apply` function to compute these column means in parallel.

```
system.time( apply( X, 1, mean ) )
   user  system elapsed
  0.002   0.000   0.002
```

This is much faster because R is built and optimised to work with vectors as much as possible. It is good practice to avoid using for loops whenever it is possible.

Exercises

1. Consider computing the matrix product ABu where A and B are 1000×1000 matrices and $u \in \mathbb{R}^{1000}$. The R code below shows that two approaches for computing this product take very different times:

```
> A <- matrix( rnorm( 1000 * 1000), nrow = 1000, ncol = 1000)
> B <- matrix( rnorm( 1000 * 1000), nrow = 1000, ncol = 1000)
> u <- rnorm( 1000 )
> system.time( Z <- A %*% B %*% u )
   user  system elapsed
  0.315   0.005   0.320
> system.time( Z <- A %*% (B %*% u) )
   user  system elapsed
  0.003   0.001   0.003
```

   What is the reason for this difference?

2. Write a function that generates an $n \times p$ design matrix X and B copies of a response vector Y from the linear model:
$$Y^{(i)} = X\beta + \epsilon^{(i)}.$$
   for $i \in \{1, \ldots, B\}$. The function should accept as arguments: the sample size n, the number of covariates p, the number of simulations B, the coefficient vector β, and a function that generates the noise variables, e.g. `rnorm`. This function should return a list that contains the design matrix as one component, and an $n \times B$ matrix Y with columns $Y^{(i)}$.

3. Write a function called `LinMod` that takes the simulated datasets from the previous function, and returns a list containing the components:
   (a) The estimated coefficients $\hat{\beta} = (X^T X)^{-1} X^T Y$;
   (b) The fitted values $\hat{Y} = X\hat{\beta}$;
   (c) The estimated noise variance $\tilde{\sigma}^{(i)} = (Y^{(i)} - \hat{Y}^{(i)})^2/(n-p)$; and
   (d) The matrix of t-statistics for each component $T_j^{(i)} = \hat{\beta}_j^{(i)} / \sqrt{(\tilde{\sigma}^{(i)})^2 \{(X^T X)^{-1}\}_{jj}}$.

   Note that each of these should be a matrix, with each column containing the values for each simulation and either 1, p, or n rows.

4. If X has full rank, $\beta_j = 0$, and $\epsilon \sim N(0, \sigma^2 I_n)$ then we know that the t-statistics have $T_j^{(i)} \sim t_{n-p}$, and that their squares will have an F-distribution on the appropriate degrees of freedom (that you should determine). If we wish to test these assumptions, we can plot the quantiles of these F-statistics against their theoretical quantiles:

```
qqplot_F <- function(f_stat, df1, df2, conf_level = 0.95) {
    f_stat <- sort(f_stat)
    B <- length(f_stat)
    theoretical_quantiles <- qf((1:B) / (B + 1), df1, df2)
    plot(theoretical_quantiles, f_stat)
    abline(0, 1, col = "red")
    return(mean(f_stat <= qf(conf_level, df1, df2)))
}
```

   What does these function return? Examine what happens to these Q-Q plots of the F-statistic vector $(T_j^{(i)})^2$ for a fixed j if we were to use alternative error distributions, e.g. `rcauchy`, and if we were to change β_j to some non zero value.

5. Modify your `LinMod` function so that it also accepts a vector G_0 that specifies a subset of the variables of X, and now also returns the $1 \times B$ matrix of F-statistics:

$$F^{(i)} = \frac{\frac{1}{p-p_0}\|(P - P_0)Y^{(i)}\|^2}{\frac{1}{n-p}\|(I - P)Y^{(i)}\|^2} \tag{1}$$

where $P = X(X^TX)^{-1}X^T$ and $P_0 = X_0(X_0^TX_0)^{-1}X_0^T$ (where X_0 is the matrix formed by the subset of columns of X) as defined in lectures. With $\beta = 0$ compare these simulated F statistics with the Q-Q plot.

6. R has a built in function `lm` that does much of the same computation as the `LinMod` function we have written. Using the first simulation of the dataset you generated (which we will call `Y1`), examine the output of calling:

```
summary( lm( Y1 ~ X ) )
```