Lab on Functional Programming and R Notebooks

R markdown

In this lab, we will run experiments using R notebook, which is a powerful programming- and note-taking tool that allows seamless integration between code and documentation. Here is an example:

```r
print("hello world!")
```

```
## [1] "hello world!"
```

You may have seen Python notebook in TB1. R notebook is exactly the same idea, but designed for R programming language. Click the "green arrow" to run your code.

Markdown is a popular scripting language that allows you to write **formatted text** in *plain text editors* (such as notepad) and render it later. You can switch between the source code and visualization using the "source" and "visual" in the toolbar and see the differences.

Targets for today

In today's lab, you will apply **functional programming** to code the pairwise distance function that we have encountered before many times. Specifically, given matrices $A \in \mathbb{R}^{n_1 \times m}$ and $B \in \mathbb{R}^{n_2 \times m}$, we would like to construct a matrix $D \in \mathbb{R}^{n_1 \times n_2}$, where $D_{i,j} = \text{dist}(A_{[i,]}, B_{[j,]})$, $A_{[i,]}$ is i-th row of A and

$$\text{dist}(a, b) = \sqrt{\sum_i (a_i - b_i)^2}.$$

Preparation

Before you start, I strongly recommend you review this week's lecture slides on list and functional programming.

Write code below, generating two random matrices A and B with 3 rows and 2 columns which are both filled with observations from standard normal distribution (Hint: `?rnorm`).

```r
#type your code here and run it by clicking the green arrow on the top-right corner.

A <- matrix(rnorm(3*2), 3,2)
B <- matrix(rnorm(3*2), 3,2)
```

Copy `pdist4` function from the previous lab (solution available online), and compute the distance matrix D using A and B. **NOTE**: in a previous lab we had $B \in \mathbb{R}^{m \times n_2}$ rather than $B \in \mathbb{R}^{n_2 \times m}$ as here. You will need to modify `pdist4` to take this into account.

```r
# Copy and modify the pdist4 code here
# fully vectorised version
pdist4 <- function(A,B){
  B <- t(B)  # Easy solution: traspose B upfront.
  nA <- nrow(A)
  nB <- ncol(B)
  o1 <- matrix(1, 1, nB)
```

```r
  o2 <- matrix(1, nA, 1)

  D <- sqrt( rowSums(A^2) %*% o1 - 2 * A%*%B + o2%*%t(colSums(B^2)) )
  return(D)
}

# type your code here
pdist4(A,B)
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.2896134 0.5671553 0.8748157
## [2,] 2.6592941 2.0713046 1.8478147
## [3,] 2.9541860 2.3141853 2.0268011
```

```r
# Optional checking whether our code is correct!
M <- matrix(NA, 3, 3)
for(ii in 1:nrow(A)){
  for(jj in 1:nrow(B)){
  M[ii, jj] <- sqrt(sum((A[ii, ]-B[jj, ])^2))
  }
}
M - pdist4(A,B) # Looks right!
```

```
##               [,1]         [,2]          [,3]
## [1,] -1.665335e-16 0.000000e+00 -1.110223e-16
## [2,]  0.000000e+00 0.000000e+00 -4.440892e-16
## [3,] -4.440892e-16 4.440892e-16  0.000000e+00
```

Pairwise Distance

We already know how to write vectorized code for computing D. Now let us inspect at this computation from a functional programming point of view.

- Given two data matrices $A \in \mathbb{R}^{n_1 \times m}$ and $B \in \mathbb{R}^{n_2 \times m}$,
- Fix k, and compute$(A_{[i,k]} - B_{[j,k]})^2$ for all i, j.
- Store the outcome as the i, j-th element of a matrix D_k.
- Apply the above computation for all k.
- $D := \sqrt{\sum_k D_k}$

Task 1

Write a function `diff2` that takes two scalars a, b, and compute $(a - b)^2$:

```r
#type your code here
diff2 <- function(a, b){
  return( (a-b)^2 )
}
```

`outer(X,Y,FUN)` applies function `FUN` two all pairs of elements in `X` and `Y`. In other words, it computes a matrix D, $D_{i,j} = \text{fun}(X_i, Y_j), \forall i, j$. For example,

```r
a <- c(1,2,3,4)
b <- c(1,2,3,4)

a_plus_b <- function(a,b){
```

```r
  return (a+b)
}

# function as an input variable
# outer applies the operation a_plus_b to each pair of elements in a and b
outer(a, b, a_plus_b)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    3    4    5
## [2,]    3    4    5    6
## [3,]    4    5    6    7
## [4,]    5    6    7    8
```

```r
# what is the i,j-th element in the matrix below?
```

Now use `outer` and `diff2`, compute a 3 by 3 matrix $D^{(1)}$, $D^{(1)}_{i,j} = \text{diff2}(A_{[i,1]}, B_{[j,1]})$.

```r
#type your code here

D1 <- outer(A[,1], B[,1], diff2)
D1
```

```
##             [,1]      [,2]      [,3]
## [1,] 0.004469899 0.2995542 0.6770899
## [2,] 1.446295527 0.5215181 0.1994720
## [3,] 3.871277548 2.2114607 1.4678747
```

Task 2

We would like to repeat the above operation for all k. We can use a for loop but, in functional programming, we tend to think we "apply" a data operation to **the entire dataset** rather than create loops that iterate over each piece of our dataset.

This is exactly what we did above: We applied `diff2` to all pairs of i, j.

Let us create another function `D_k`, that takes one integer input k and compute D_k using `outer` (wrap the code you have just written in a function)

```r
# type your code here
D_k <- function(k){
  return(outer(A[,k], B[,k], diff2))
}
```

Now apply this function to a list of numbers (hint: `?lapply`). The list contains integers from 1 to m, where m is 2 in this toy case.

```r
# type your code here
res <- lapply(list(1,2), D_k)
```

You should get a list of matrices whose k -th element is D_k.

Task 3

Now, all that is left is summing D_k together and we are done! Can we use a for loop to sum all D_k matrices together?

In Functional Programming, we tend to think our program is a huge pipeline that applies functions repeatedly to our data: `op3(op2(op1(data)))...` (recall the "data pipeline" in the lecture slides)

We can rewrite the summation of D_k using this paradigm:

$\sum_{k=1...4} D_k$ is the same as

`sum2(sum2(sum2(D_1,D_2),D_3),D_4)`, where `sum2(X,Y)` sums two matrices `X` and `Y` together. Write a function `sum2` that sums over two matrices

```r
# type your code here

sum2 <- function(a,b){
  return(a+b)
}
```

Now we need to apply this function to the list of D_k recursively. This can be done using a function called `Reduce`. `Reduce(f,x)` applies `f` to a list `x` recursively. In other words, it computes

`f(f(f(x_1, x_2),x_3),x_4)...`

```r
# type your code here

res2 <- Reduce(sum2, res)
```

Square root the outcome.

```r
# type your code here

D <- sqrt(res2)
D
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.2896134 0.5671553 0.8748157
## [2,] 2.6592941 2.0713046 1.8478147
## [3,] 2.9541860 2.3141853 2.0268011
```

Check, is this the same result you obtained using `pdist4`?

```r
# Your checking code here

D - pdist4(A,B)
```

```
##               [,1]         [,2]          [,3]
## [1,] -1.665335e-16 0.000000e+00 -1.110223e-16
## [2,]  0.000000e+00 0.000000e+00 -4.440892e-16
## [3,] -4.440892e-16 4.440892e-16  0.000000e+00
```

Task 4

`lapply(l, fun)` applies a function `fun` to a list `l`.

There is a sibling function called `apply(A, MARGIN, fun)` which applies the function `fun` to a matrix along a certain margin. See `?apply`.

Setting `MARGIN` to 1 means you are applying `fun` to each row of the matrix.

Setting `MARGIN` to 2 means you are applying `func` to each column of the matrix.

`apply` applies `fun` to rows/columns of `A`, return a vector containing the results.

```r
X <- matrix(c(1,2,3,4), 2, 2)
X
```

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

```r
# compute the average of a vector
average <-function(r) {
  return(sum(r)/length(r))
}

# compute row average
apply(X, 1, average)
```

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

Now, given the `D` matrix you obtained. Could you sort the elements of each row of `D`, in ascending order?

Hint: `sort` function, takes in a vector, and sort them according to the ascending order. `?sort`

```r
sort(c(4,3,2,1))
```

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

```r
# your code here

D
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.2896134 0.5671553 0.8748157
## [2,] 2.6592941 2.0713046 1.8478147
## [3,] 2.9541860 2.3141853 2.0268011
```

```r
t(apply(D, 1, sort))
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.2896134 0.5671553 0.8748157
## [2,] 1.8478147 2.0713046 2.6592941
## [3,] 2.0268011 2.3141853 2.9541860
```