STAT 2005 – Programming Languages for Statistics

Tutorial 6 Matrix, R packages

2020

LIU Ran

Department of Statistics, The Chinese University of Hong Kong

1 Matrix

1.1 Vector And Column Matrix

The difference between a vector and a column matrix:

```
# a vector is not a matrix
> a = 1:4
# we can use is.matrix to check whether or not the object is a matrix
> is.matrix(a)
[1] FALSE

# If we transform the vector to be a matrix, it will be a column matrix instead of a row matrix
> (b = as.matrix(a))
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
> is.matrix(b)
[1] TRUE

# we can also use t transpose function to get a row matrix
> (c = t(a))
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
> is.matrix(c)
[1] TRUE

# t(t()) is equivalent to as.matrix() for a vector
> (d = t(t(a)))
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
> is.matrix(d)
[1] TRUE
```

Remark 1.1. Vectors perform like column matrices but are not exactly the same. Be careful when doing matrix algebra using vectors.

1.2 diag

diag is a generic function like what I told you in the last tutorial. It will have different effects(methods) on different objects.

```
# input a vector
> diag(1:4)
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
[4,]    0    0    0    4

# input a scalar
> diag(4)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1

> (a = matrix(1:12, ncol=4))
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
# input a matrix
# the matrix may not be an n*n square matrix
> diag(a)
[1] 1 5 9

# use the submatrix with the min dimension
> (b = matrix(1:12, nrow=4))
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> diag(b)
[1]  1  6 11

# replacement function
> diag(b)<-1
```

1.3 Matrix Algebra

```
# use a vector to create a matrix
> a = 1:4
# if there are not enough elements, it will automatically replicate this vector several times to
    match the shape of the matrix
> (b = matrix(a, nrow=3, ncol=4))
     [,1] [,2] [,3] [,4]
[1,]    1    4    3    2
[2,]    2    1    4    3
[3,]    3    2    1    4

# each element will plus the scalar
> b+1
     [,1] [,2] [,3] [,4]
[1,]    2    5    4    3
[2,]    3    2    5    4
[3,]    4    3    2    5

# also automatically replicate
> (c = matrix(2, nrow=3, ncol=4))
     [,1] [,2] [,3] [,4]
[1,]    2    2    2    2
[2,]    2    2    2    2
[3,]    2    2    2    2

# same shape matrix
# the result will be the summation on each position
> b+c
     [,1] [,2] [,3] [,4]
[1,]    3    6    5    4
[2,]    4    3    6    5
[3,]    5    4    3    6

# the result will be the product on each position
> b*c
     [,1] [,2] [,3] [,4]
[1,]    2    8    6    4
[2,]    4    2    8    6
[3,]    6    4    2    8

# ncol(b) is 4; nrow(c) is 3; not match for the matrix multiplication
> b%*%c
```

```
Error in b %*% c : non-conformable arguments
> b%*%t(c)
     [,1] [,2] [,3]
[1,]   20   20   20
[2,]   20   20   20
[3,]   20   20   20

# matrix + vector: vector will be transformed into the same shape matrix automatically first
> b+a  # b+matrix(a, nrow=3, ncol=4)
     [,1] [,2] [,3] [,4]
[1,]    2    8    6    4
[2,]    4    2    8    6
[3,]    6    4    2    8

# matrix * vector: vector will be transformed into the same shape matrix automatically first
> b*a # b*matrix(a, nrow=3, ncol=4)
     [,1] [,2] [,3] [,4]
[1,]    1   16    9    4
[2,]    4    1   16    9
[3,]    9    4    1   16

# multiply a on the right
# regard the vector as a column matrix
# 3*4 matrix times 4*1 matrix = 3*1 matrix
> b%*%a
     [,1]
[1,]   26
[2,]   28
[3,]   26

# multiply a on the left
# regard the vector as a row matrix
> a%*%t(b)
     [,1] [,2] [,3]
[1,]   26   28   26

# create a matrix which has different dimensions
> (d = matrix(1:16, ncol=4))
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16

#  return errors with different dimension matrixes
> b+d
Error in b + d : non-conformable arrays
> b*d
Error in b * d : non-conformable arrays

# use the det function to calculate the determinant of a matrix
> det(d)
[1] 0
# d is a singular matrix, there is no inverse matrix for it
> solve(d) # try to get the inverse matrix
Error in solve.default(d) :
  Lapack routine dgesv: system is exactly singular: U[3,3] = 0
> (e = matrix(1:4, ncol=2))
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> det(e)
[1] -2
> solve(e)
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

Remark 1.2. Check dimensions before doing matrix algebra.

2 Mathematical calculation

2.1 Integration

You can use **integrate** command to get the integral of the function you input in a specific interval.

```
> integrate(dnorm, -1.96, 1.96)
0.9500042 with absolute error < 1e-11

# you can also set the infinity value in the interval
> integrate(dnorm, -Inf, Inf)
1 with absolute error < 9.4e-05

# integrate must accept vectorized function
# for example, a function which is not vectorized
> f <- function(x) 2.0 # just return the value 2.0
> integrate(f, 0, 1)
Error in integrate(f, 0, 1) :
  evaluation of function gave a result of wrong length
> integrate(Vectorize(f), 0, 1)  ## correct
2 with absolute error < 2.2e-14
```

Remark 2.1. Because we use numeral calculations instead of the theoretical values, it will show the absolute error here.

2.2 Derivatives

Compute derivatives of simple expressions:**D**

```
# define a expression
> (f = expression(x^2))
expression(x^2)
> D(f,"x") # it will show the expression for derivatives
2 * x

# use eval command to calculate the function value at point x=5
> x = 5 # assign a value to x
> eval(D(f,"x")) # use eval command to operate on the expression
[1] 10
```

Remark 2.2. Because here we want to get the theoretical formula of derivatives. We cannot use the **function** command to define the expression here, which means we cannot use `f<-function(x)x**2`.

Remark 2.3. **function** must do something(such as calculating). You must have the input, function body and the output. **expression** is just the expression. It does nothing and just show the structure of your formula.

3 Optimization

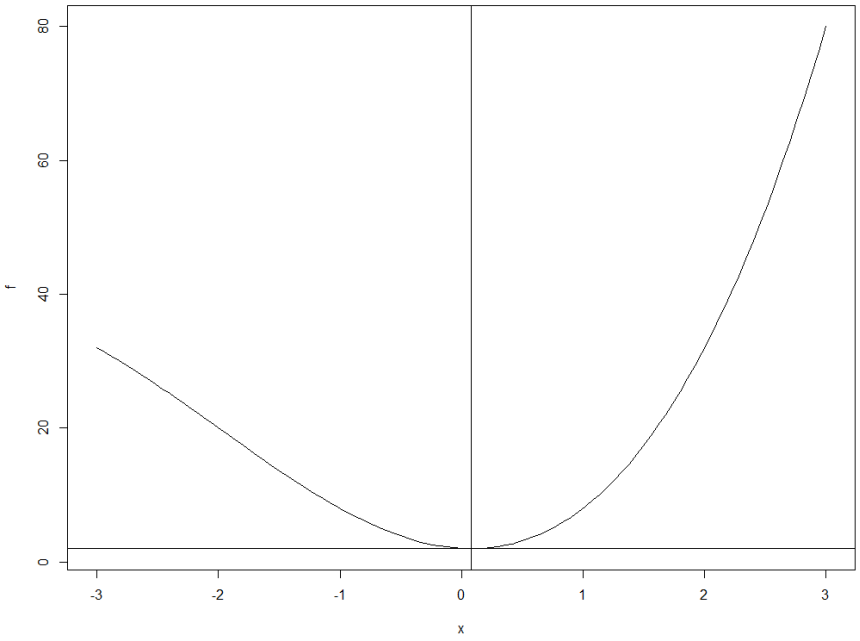
The function **optimize** searches the interval from lower to upper for a minimum of the function f with respect to its first argument.

```
> f = function(x){x^3+6*x^2-x+2}
> plot(f, -3, 3)
> (fit = optimize(f, c(-2,2), tol=0.00001))
$minimum # x coordinate
[1] 0.08166439

$objective # the function value or y coordinate
[1] 1.958895

# draw the point on the figure
> abline(v=fit$minimum)
> abline(h=fit$objective)
```

```
13
14 # we can input other arguments required for the function
15 # for example, two dimensional function f(x,y)
16 > f = function(x, y){x^3+6*x^2-x+2+y^2}
17 # optimize the f(x,y=2)
18 # y^2 = 4
19 > optimize(f, c(-2,2), tol=0.00001, y=2)
20 $minimum
21 [1] 0.08166439
22
23 $objective # these two objectives will differ by 4
24 [1] 5.958895
```



4 An exercise about the determinant

We can use the `det()` command to calculate the Determinant of a matrix.

In the case of a 2 × 2 matrix the determinant may be defined as

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc.$$

Similarly, for a 3 × 3 matrix A, its determinant is

$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$
$$= aei + bfg + cdh - ceg - bdi - afh.$$

Each determinant of a 2 × 2 matrix in this equation is called a minor of the matrix A. This procedure can be extended to give a recursive definition for the determinant of an n × n matrix. After the expansion, the dimension will reduce.

Therefore, we can use recursive code to calculate the determinant by ourselves.

```
> set.seed(2005)
>
> my_det<-function(A){
+
+   # store the sign of one part
+   indicator = 1
+   result = 0
+
+   # stopping criterion
+   # if the number of the column is equal to 2, we end the recursion
+   if(ncol(A)==2){
+     return(A[1,1]*A[2,2]-A[1,2]*A[2,1])
+   }
+
+   # calculate each part by the column order
+   for (j in 1:ncol(A)) {
+
+     # get the submatrix(minor)
+     temp_A = A[-1,-j]
+     result = result + indicator*A[1,j]*my_det(temp_A)
+
+     # change the sign of the indicator
+     # staggered sign
+     indicator = -indicator
+   }
+   return(result)
+ }
>
> n = 4
> A = matrix(runif(n**2),nrow = n)
> my_det(A)
[1] 0.05846946
> det(A)
[1] 0.05846946
```

5 R package: dplyr(optional)

We can use **require** or **library** to load a specific R package. When you try to load a package which doesn't exist, **require** will give warning messages and return FALSE; **library** will return error and terminate the process.

```
> require('sssss')
Warning message:
In library(package, lib.loc = lib.loc, character.only = TRUE, logical.return = TRUE,  :
> library('sssss')
Error in library("sssss") :
> a = require('sssss')
> a
[1] FALSE
```

dplyr is a set of functions designed to enable dataframe manipulation in an intuitive, user-friendly way. Here are some main functions:

1. **filter**: filters out rows according to some conditions.
2. **arrange**: reorders rows according to some conditions
3. **select**: selects a subset of columns
4. **mutate**: adds a new column as a function of existing columns

%>% pipeline operator: All of the dplyr functions take a data frame as the first argument. Rather than forcing the user to either save intermediate objects or nest functions, dplyr provides the **%>%** operator. x **%>%** f(y) turns into f(x, y) so the result from one step is then "piped" into the next step. You can use the pipe to rewrite multiple operations that you can read left-to-right, top-to-bottom.

```
> # load the package
> require('dplyr')
> # Use the built-in data set iris
> str(iris)
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
>
> # filter out rows: Species equal to virginica and Sepal.Length greater than 7.5
> iris %>%
+   filter(Species == 'virginica', Sepal.Length > 7.5)
  Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
1          7.6         3.0          6.6         2.1 virginica
2          7.7         3.8          6.7         2.2 virginica
3          7.7         2.6          6.9         2.3 virginica
4          7.7         2.8          6.7         2.0 virginica
5          7.9         3.8          6.4         2.0 virginica
6          7.7         3.0          6.1         2.3 virginica
>
> filter(iris, Species == 'virginica', Sepal.Length > 7.5)
  Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
1          7.6         3.0          6.6         2.1 virginica
2          7.7         3.8          6.7         2.2 virginica
3          7.7         2.6          6.9         2.3 virginica
4          7.7         2.8          6.7         2.0 virginica
5          7.9         3.8          6.4         2.0 virginica
6          7.7         3.0          6.1         2.3 virginica


# after filtering, sort the data by the column Sepal.Length and Sepal.Width
# first, compare the Sepal.Length, if they are equal, compare the Sepal.Width
# And finally create a new column which is the average value of two lengths
> iris %>%
+   filter(Species == 'virginica', Sepal.Length > 7.5) %>%
+   arrange(Sepal.Length, Sepal.Width) %>%
+   mutate(ave_length = (Sepal.Length+Petal.Length)/2)

  Sepal.Length Sepal.Width Petal.Length Petal.Width   Species ave_length
1          7.6         3.0          6.6         2.1 virginica       7.10
2          7.7         2.6          6.9         2.3 virginica       7.30
3          7.7         2.8          6.7         2.0 virginica       7.20
4          7.7         3.0          6.1         2.3 virginica       6.90
5          7.7         3.8          6.7         2.2 virginica       7.20
6          7.9         3.8          6.4         2.0 virginica       7.15

# If you don't use the pipeline operator, it is not clear enthogh.
> temp_data1 = filter(iris, Species == 'virginica', Sepal.Length > 7.5)
> temp_data2 = arrange(temp_data1,Sepal.Length, Sepal.Width)
> temp_data3 = mutate(temp_data2,ave_length = (Sepal.Length+Petal.Length)/2)
```

Remark 5.1. Tutorials for **dplyr** and the vignettes.