DNSC 6279 Section 11 Spring 2020 Homework 1

Ziwei Li

Remark. This homework aims to introduce you to the basics in **R**, which would be the main software we shall work on throughout this course.

Instruction. Download the whole folder for this homework. Open the **RMarkdown** document with surfix ".rmd" via **RStudio**. Click `Knit` to create a PDF document (remember to install the necessary packages in your **R**). By removing the `results='hide'` and `fig.keep='none'` options in the code chunks, the code outputs and the plots would display in the created file. For more information about the **RStudio**, refer to the section **Getting Started**; about the **RMarkdown**, refer to the online tutorial (http://rmarkdown.rstudio.com/lesson-1.html) and the online manual of `knitr` (https://yihui.name/knitr/) by Yihui Xie from **RStudio, Inc.**

> **Please turn off the display of example code chunks (by specifying `include=FALSE`), complete the exercise code chunks (remember to turn on the `eval` option), fill in your name and create a PDF document, then print and submit it.**

Getting Started

R is one of the most common coding language in data analysis nowadays. It's a free, open-source and powerful software environment for statistical computing and graphics. A nice description of **R** can be found in the course website (http://data.princeton.edu/R/) of German Rodriguez from Princeton. To download and install **R**, click into the CRAN (Comprehensive R Archive Network) Mirrors (https://cran.r-project.org/mirrors.html) and choose a URL that applies to you. To run with **R**, click "RGui.exe" in Windows system or "R.app" in Mac OS.

The console of **R** is not the most friendly interface to work with, as compared to the more handful editor **RStudio**. You can download and install it via its website (https://www.rstudio.com/products/rstudio/download/) and choose the free version of **RStudio Desktop**. Note that when you run with **RStudio**, the mirror of **R** should have been installed in your system, even though you are not accessing to it directly.

The coming sections lead you to the essentials of **R**. To explore more about how **R** works, please refer to (Dalgaard 2008, @venables2017) and the tutorial (http://data.princeton.edu/R/) of German Rodriguez.

Basics in R

R is an object-oriented language. Hence the "data" we work on are formatted as a particular object that meets some structural requirements (subjected to a particular class). Thus one should first understand which class of

object he/she has on hand, and then figures out the applicable operations on it.

In a hierarchical manner, the more advanced class consists of ingredients from more fundamental classes. Vectors, matrices, lists, data frames and factors are the most commonly used fundamental classes in data analysis.

Vectors and Matrices

Vector is a collection of “data” that share the same type (numeric, character, logic or NULL). And matrix arranges “data” of the same type in two dimensions. Note that there doesn’t exist a “scalar” object, which would be treated as a vector of length 1.

Create a Vector

The combining function `c( )` can be used to manually create a vector in **R**. When using the `c( )` function, numbers are entered as a list with commas between each new entry. For example, `x <- c(1, 2)` creates a vector and assigns it to the variable `x`.

To create a vector that repeats n times, we can use the replication function `rep( , n)`. For example, a vector of five TRUE’s can be obtained by `x <- rep(TRUE, 5)`.

Finally, we can create a consecutive sequence of numbers using the sequence generating function `seq(from = , to = , by = )`. Here, the `from`, `to` and `by` arguments specify where the sequence begins, ends, and by how much the sequence increments. For example, the vector $(2, 4, 6, 8)$ can be obtained using `x <- seq(2 , 8, 2)`. A convenient operator `:` similar to `seq` also creates the consecutive sequence stepped by 1 or -1. Try `1:4` and `4:1`.

For more information, the commands `?c`, `?rep` and `?seq` access to the online **R** documents for help.

Exercise 1. *(5 pt)* Using the `c`, `rep` or `seq` commands, create the following 6 vectors:

$x1 = (2, .5, 4, 2);$

$x2 = (2, .5, 4, 2, 1, 1, 1, 1);$

$x3 = (1, 0, -1, -2);$

$x4 = (\text{" Hello "}, \text{" "}, \text{" World, "}, \text{"! "}, \text{" Hello World! "});$

Hint. Try `paste` function.

$x5 = (\text{TRUE}, \text{TRUE}, \text{NA}, \text{FALSE});$

Hint. Check `?NA` and `class(NA)` to learn more about the missing value object `NA`.

$x6 = (1, 2, 1, 2, 1, 1, 2, 2).$

```
x1 = c(2,0.5,4,2)
x2 = c(x1,rep(1,4))
x3 = seq(1,-2,-1)
x4 = c("Hello","","World","!",paste("Hello","","World","!"))
x5 = c(TRUE,TRUE,NA,FALSE)
x6 = c(rep(c(1,2),2),rep(1,2),rep(2,2))
x1
```

```
## [1] 2.0 0.5 4.0 2.0
```

```
x2
```

```
## [1] 2.0 0.5 4.0 2.0 1.0 1.0 1.0 1.0
```

```
x3
```

```
## [1]  1  0 -1 -2
```

```
x4
```

```
## [1] "Hello"          ""               "World"          "!"
## [5] "Hello  World !"
```

```
x5
```

```
## [1]  TRUE  TRUE    NA FALSE
```

```
x6
```

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

Create a Matrix

An m-by-n matrix can be created by the command `matrix( , m, n)` where the first argument admits a vector with length compatible with the matrix dimensions. For example, `x <- matrix(1:4, 2, 2)` creates a 2-by-2 matrix that arranges the vector (1, 2, 3, 4) by column. To arrange the vector by row, specify the `byrow` option like this: `x <- matrix(1:4, 2, 2, byrow = TRUE)`.

Moreover, the command binding vectors/matrices by row `rbind` and by column `cbind` are also useful. Check **R** documents for their usages.

Exercise 2. *(5 pt)* Using `matrix` , `rbind` and `cbind` , create

$$\mathbf{X} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & -1 & -2 \\ 2 & .5 & 4 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

```
z1 = c(1,2,3,4)
z2 = c(1,0,-1,-2)
z3 = c(2,.5,4,2)
z4 = rep(1,4)
X <- matrix(c(z1,z2,z3,z4),4,4,byrow=TRUE)
X <- rbind(z1,z2,z3,z4)
X <- cbind(X[,1],X[,2],X[,3],X[,4])
X
```

```
##    [,1] [,2] [,3] [,4]
## z1    1  2.0    3    4
## z2    1  0.0   -1   -2
## z3    2  0.5    4    2
## z4    1  1.0    1    1
```

Indexing

There are three ways to extract specific components in a vector.

The second approach leads to the so called “conditional selection” technique as follows.

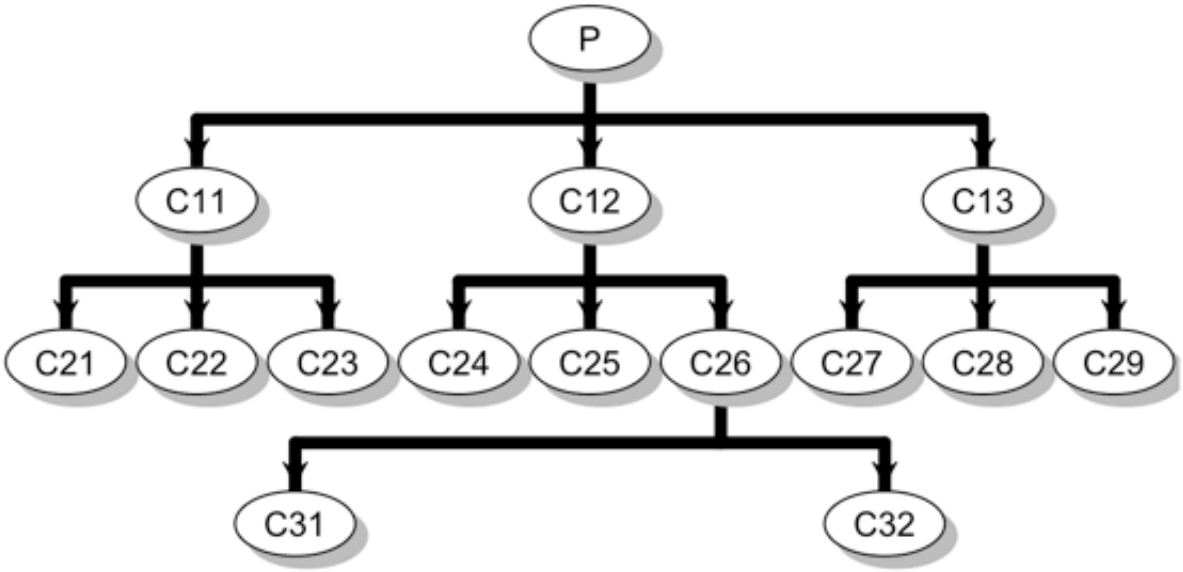
Matrix indexing follows the same manner.

List

List is a more flexible container of “data” that permits inhomogeneous types. It’s useful if you would like to encapsulate a bunch of components in an object. The `list` function explicitly specifies a list and the combining function `c` is still applicable. For example,

```
x <- list( num   = 1:4,               # "num =" specifies the name of the first comp
onent
           chac  = "hello world!",
           logic = c(TRUE,FALSE),
           nu    = NULL,
           mat   = matrix(4:1, 2, 2) ) # to the left of = specifies the component nam
e
y <- list( 1234,
           "world" )
c(x, y)
```

To extract the components in a list, one should use double bracket `[[ ]]` instead of a single bracket. If you've already specified the component names in a list, then the component names can be placed into the bracket directly. For example, `x[["logic"]]` accesses the third component of `x`. A more handful alternative is `x$logic`.



Date Frame

Inheriting from matrix and list, data.frame is a container general enough for us to study a dataset. It permits inhomogeneous data types across columns (components in a list) but forces the components of the list to be vectors of homogeneous length (so as to be columns in a matrix). For example, the following creates a score table of 3 students.

```
students <- data.frame( id      = c("001", "002", "003"), # ids are characters
                        score_A = c(95, 97, 90),          # scores are numericss
                        score_B = c(80, 75, 84))
students
```

To access the score_A of student 003, one can follow the manner in a matrix: `students[3,2]`, or that in a list: `students[[[2]][3]`, `students[["score_A"]][3]` or `students$score_A[3]`.

Exercise 3. *(5 pt)* Applying the conditional selection technique (without using `subset`), extract the record of 003 in `students`.

```
students[students['id']=='003']
```

```
## [1] "003" "90"  "84"
```

```
print(students[3,])
```

```
##    id score_A score_B
## 3 003      90      84
```

One can also create a matrix or a legitimate list first and then convert it into a data.frame as follows.

Exercise 4. *(10 pt)* Create a data.frame object to display the calendar for Jan 2018 as follows.

1) The character object `""` ; 2) the option `row.names = FALSE` in `print` function.

```
## Sun Mon Tue Wed Thu Fri Sat
##      NY   2   3   4   5   6
##   7   8   9  10  11  12  13
##  14 MLK  16  17  18  19  20
##  21  22  23  24  25  26  27
##  28  29  30  31
Day_of_week <- c("SUN","MON","TUE","WED","THU","FRI","SAT")
x <- t(matrix(0:34,7,5,byrow=FALSE))
x[[1,1]] <- ""
x[[5,5]] <- ""
x[[5,6]] <- ""
x[[5,7]] <- ""
x[[1,2]] <- "NY"
x[[3,2]] <- "MLK"
colnames(x) <- Day_of_week
x
```

```
##      SUN  MON   TUE  WED  THU  FRI  SAT
## [1,] ""   "NY"  "2"  "3"  "4"  "5"  "6"
## [2,] "7"  "8"   "9"  "10" "11" "12" "13"
## [3,] "14" "MLK" "16" "17" "18" "19" "20"
## [4,] "21" "22"  "23" "24" "25" "26" "27"
## [5,] "28" "29"  "30" "31" ""   ""   ""
```

Factors

Factor is a special data structure in **R** in representing categorical variables and facilitating the data labels and subgroups. It's basically a character vector that keeps track of its distinct values called levels. Consider the longitudinal layout of the previous score table.

```
id        <- rep(c("001","002","003"), 2)
subj      <- rep(c("A","B"), each = 3)
score     <- c(95, 97, 90, 80, 75, 84)
students3 <- data.frame(id, subj, score)  # try cbind(id, subj, score) to see the dif
ference

# students3$id and students3$subj are automatically formatted as factors
class(students3$id)
levels(students3$id)

class(students3$subj)
levels(students3$subj)

# combind student 003 with 002 via level rename
students4 <- students3    # work on a copy in case of direct modification of students
3
levels(students4$id)[3] <- "002"
levels(students4$id)
students4
```

The `factor` function applied to a character vector creates a natural factor. The `gl` and `cut` functions are also useful approaches to patterned factors and that generated from numeric variables.

Exercise 5. *(5 pt)* Create a factor variable `grade` in `students3`, where the `score` variable is divided into $[90, 100]$, $[80, 90)$ and $[0, 80)$ corresponding to A, B and C in `grade` respectively.

Hint. Functions `cut` and `transform`.

```
students5 <- students4
students5$grade <- cut(students5$score,breaks=c(0,80,90,100),labels=c("C","B","A"),ri
ght=FALSE)
students5
```

```
##    id subj score grade
## 1 001    A    95     A
## 2 002    A    97     A
## 3 002    A    90     A
## 4 001    B    80     B
## 5 002    B    75     C
## 6 002    B    84     B
```

Operations and Functions

Note that scalar operations on vectors usually apply componentwise.

- **Arithmetic operations:** `+`, `-`, `*`, `/`, `^`, `%/%` (exact division), `%%` (modulus), `sqrt()`, `exp()`, `log()`
- **Logical operations**

    - And `&`, `&&`; or `|`, `||`; not `!`
    - Comparisons: `<`, `<=`, `>`, `>=`, `==` (different from `=`), `!=`
    - Summary functions: `all()`, `any()`
- **Summary statistics:** `length`, `max`, `min`, `sum`, `prod`, `mean`, `var`, `sd`, `median`, `quantile`
- **Matrix operations**
    - Matrix multiplication: `%*%` (different from `*`)
    - Related functions: `t`, `solve`, `det`, `diag`, `eigen`, `svd`, `qr`
    - Marginal operations: `apply`
- **Factors:** `tapply` (to apply operations/functions grouped by a factor)

Exercise 6. *(5 pt)* Without using the `var` and `scale` functions, compute the sample covariance `X.var` of the data matrix `X` as in **Exercise 2.**

```
# Formula of a matrix's covariance:
# For Xij:
# if i=j:
  #COV(Xij,Xij) = variance(Xij) = (1/N-1)[(X1-Xbar)^2+(X2-Xbar)^2+...+((Xn-Xbar)^2)]
# if i != j:
  # Cov(Xij,Xab) = (1/N-1)[(X1j-mean(Xj)(X1b-mean(Xb)+(X2j-mean(Xj)(X2b-mean(Xb))+...
#   +(Xij-mean(Xj))(Xab-mean(Xb)))]
## Step 1: get the mean of each column
col1 = mean(X[,1])
col2 = mean(X[,2])
col3 = mean(X[,3])
col4 = mean(X[,4])
## step 2: get the number of columns and rows
col_num = ncol(X)
row_num = nrow(X)
## step 3: transform each column's mean to a col*col matrix
mean_mat = matrix(c(col1,col2,col3,col4),col_num,col_num,byrow=TRUE)
## step 4: calculate the difference between the original matrix and the mean matrix
difference_mat = X - mean_mat
## step 5: get the transpose matrix of difference
tran_difference_mat = t(difference_mat)
## step 6: calculate the unbiased estimator
unbia_est = 1/(col_num-1)
## step 7: calculate the covariance of this matrix
X.var = unbia_est * tran_difference_mat %*% difference_mat
X.var
```

```
##        [,1]       [,2]      [,3]  [,4]
## [1,]  0.250 -0.1250000 0.7500000 0.250
## [2,] -0.125  0.7291667 0.9583333 1.875
## [3,]  0.750  0.9583333 4.9166667 4.750
## [4,]  0.250  1.8750000 4.7500000 6.250
```

```
## step 8: check the results using var(X) to see if they are correct
var(X)
```

```
##        [,1]       [,2]      [,3]  [,4]
## [1,]  0.250 -0.1250000 0.7500000 0.250
## [2,] -0.125  0.7291667 0.9583333 1.875
## [3,]  0.750  0.9583333 4.9166667 4.750
## [4,]  0.250  1.8750000 4.7500000 6.250
```

Exercise 7. *(10 pt)* Create a variable `score.mean` in `students3` , taking value as the mean score among students who have the same `subj` value.

```
students3$score.mean <- ifelse(students3$subj=="A", with(students3, mean(score[subj =
= "A"])), with(students3, mean(score[subj == "B"])))
students3
```

```
##    id subj score score.mean
## 1 001    A    95   94.00000
## 2 002    A    97   94.00000
## 3 003    A    90   94.00000
## 4 001    B    80   79.66667
## 5 002    B    75   79.66667
## 6 003    B    84   79.66667
```

Writing your own functions

It's convenient to create a user-defined **R** function, where you might encapsulate a standard, complicated or tedious procedure/algorithm in a "black box" like any built-in functions as mensioned above, so that other users might only need to care about the input and output of your function regardless the details. See the following toy example.

```
my.fun <- function(x, y)
{
  # This function takes x and y as input
  # It returns a list that contains the mean of x, y respectively and their differenc
e
  mean.x    <- mean(x)
  mean.y    <- mean(y)
  mean.diff <- mean.x - mean.y
  output    <- list(mean_of_x = mean.x,
                    mean_of_y = mean.y,
                    mean_diff = mean.diff)
  return(output)
}
x <- runif(50, 0, 1)  # simulate 50 numbers from U[0,1]
y <- runif(50, 0, 3)  # simulate 50 numbers from U[0,3]
my.fun(x, y)
```

Flow Control

To help you write down your own **R** programs, the following examples familiarize you with the conditional statements and loops.

Conditional Statements

Loops

Exercise 8. *(15 pt)* Write a function `bisect(f, lower, upper, tol = 1e-6)` to find the root of the univariate function `f` on the interval [`lower` , `upper`] with precision tolerance $\leq$ `tol` (defaulted to be 10^{-6}) via bisection, which returns a list consisting of `root` , `f.root` (`f` evaluated at `root`), `iter` (number of iterations) and `estim.prec` (estimated precision). Apply it to the function

$$f(x) = x^3 - x - 1$$

on $[1, 2]$ with precision tolerance 10^{-6}. Compare it with the built-in function `uniroot` .

```
bisect <- function(f, lower, upper, tol){
  a = lower
  b = upper
  iter = 0
  if(f(lower)*f(upper)>0){
    print("There is no root")
  }else{
    repeat{
  if(f(lower)*f(upper)<0){
    if(abs(lower-upper)<tol){    #If the absolute difference is less than tol, loop w
ill stop.
      break
    }else{m=(lower+upper)/2
    if(f(lower)*f(m)<0){
      upper = m
    }else{
      lower = m
    }}
    iter = iter+1
    estim.prec = (b-a)/(2^iter)
  }
    }
  }
  list(root=(lower+upper)/2,f.root=f(m),iter=iter,estim.prec=estim.prec)
}
bisect(function(x) x^3 - x - 1,1,2,tol = 1e-6)
```

```
## $root
## [1] 1.324718
##
## $f.root
## [1] -1.857576e-06
##
## $iter
## [1] 20
##
## $estim.prec
## [1] 9.536743e-07
```

```
#Using uniroot function to check if the results are correct
uniroot(function(x) x^3 - x - 1,c(1,2),tol = 1e-7)
```

```
## $root
## [1] 1.324718
##
## $f.root
## [1] 1.19238e-13
##
## $iter
## [1] 8
##
## $init.it
## [1] NA
##
## $estim.prec
## [1] 5e-08
```

Input/Output

`scan` and `read.table` are two main functions to read data. The main difference of them lies in that `scan` reads one component (also called "field") at a time but `read.table` reads one line at a time. Hence `read.table` requires the data to be well-structured as a table so as to create a data.frame in **R** automatically, while `scan` can be flexible but might require efforts in manipulating data after reading. Their usages are quite similar. One should pay attention to the frequently used options `file` , `header` , `sep` , `dec` , `skip` , `nmax` , `nlines` and `nrows` in thier **R** documents.

`write.table` is a converse function against `read.table` , while their usages are almost identical. To get familiar with thier features, explore in the next exercise.

Exercise 9. *(15 pt)* Without using `cor` , compute the sample correlation matrix `X.cor` from `X.var` in **Exercise 6.** Output `X.cor` to a text file "X_cor.txt" which displays as in Table . Then input "X_cor.txt" in **R** and reproduce the correlation matrix `X.cor1` .

Hint. 1) Functions `round` and `lower.tri` ; 2) the `NA` trick; 3) options `sep = "\t", col.names = NA` .

```
##       V1    V2   V3   V4
## V1  1.00 -0.29 0.68 0.20
## V2 -0.29  1.00 0.51 0.88
## V3  0.68  0.51 1.00 0.86
## V4  0.20  0.88 0.86 1.00
sd1 = sd(X[,1])
sd2 = sd(X[,2])
sd3 = sd(X[,3])
sd4 = sd(X[,4])
sdv = c(sd1,sd2,sd3,sd4)
mat = c()
for (i in sdv){
  for (j in sdv){
    d = i*j
      mat = append(mat,d)
  }
}
X.cor1 <- X.var/matrix(mat,4,4,byrow=FALSE)
cor2 = round(X.cor1,2)
cor2[lower.tri(cor2,diag=TRUE)] <- NA
cor2
```

```
##      [,1]  [,2] [,3] [,4]
## [1,]   NA -0.29 0.68 0.20
## [2,]   NA    NA 0.51 0.88
## [3,]   NA    NA   NA 0.86
## [4,]   NA    NA   NA   NA
```

```
write.table(cor2,file="X_cor.txt",sep='\t',col.names=NA)
cor2.2 <- read.table("X_cor.txt",header=TRUE)
cor2.2
```

```
##   X V1    V2   V3   V4
## 1 1 NA -0.29 0.68 0.20
## 2 2 NA    NA 0.51 0.88
## 3 3 NA    NA   NA 0.86
## 4 4 NA    NA   NA   NA
```

To read inline, one can specify `file = stdin()` (or omitted in `scan` function). In that case, it reads from console that the user can input line-by-line, or from the subsequent lines in a program script, until an empty line is read. However, such trick is NOT compatible in **RMarkdown**.

If you have your data stored in another format, *e.g.* EXCEL or SAS dataset, then you can output it as a CSV file and read in **R** via `read.csv` function (almost identical to `read.table`).

Probability and Distributions

This section explores how to create “randomness” in **R** and obtain probabilistic quantities.

Discrete Random Sampling

Much of the earliest work in probability theory starts with random sampling, *e.g.* from a well-shuffled pack of cards or a well-stirred urn. The `sample` function applies such procedure to a vector in **R**. Learn more from the **R** documents.

The following exercise means to create a five-fold cross-validating sets, which would be the starting point to assess the performance of a learned machine in, for example, classification errors.

Exercise 10. *(10 pt)* `iris` is a built-in dataset in **R**. Check `?iris` for more information. Randomly divide `iris` into five subsets `iris1` to `iris5` stratified to `iris$Species` (namely, the proportion of `iris$Species` among different levels remains identical across all subsets).

```
# Step 1: Create ID column in iris_new dataset
iris_new <- iris
iris_new$id <- seq(1,nrow(iris),1)
# Step 2: Shuffle iris_new dataset
## get number of rows in iris
rows <-  sample(nrow(iris_new))
## Shuffle dataset
iris_new <- iris_new[rows, ]
# Step 3: Create subset for each species
vigrinica <- subset(iris_new,Species=="virginica")
versicolor <- subset(iris_new,Species=="versicolor")
setosa   <- subset(iris_new,Species=="setosa")
# Step 4: Split each subsets into 5 sub_subsets
se_t <- split(setosa, rep(1:ceiling(50/10), each=10, length.out=50))
ver_s <- split(versicolor, rep(1:ceiling(50/10), each=10, length.out=50))
vig_r <- split(vigrinica, rep(1:ceiling(50/10), each=10, length.out=50))
# Step 5: Create each 5 sub_subsets
set1 <- se_t[[1]]
set2 <- se_t[[2]]
set3 <- se_t[[3]]
set4 <- se_t[[4]]
set5 <- se_t[[5]]
ver1 <- ver_s[[1]]
ver2 <- ver_s[[2]]
ver3 <- ver_s[[3]]
ver4 <- ver_s[[4]]
ver5 <- ver_s[[5]]
vig1 <- vig_r[[1]]
vig2 <- vig_r[[2]]
vig3 <- vig_r[[3]]
vig4 <- vig_r[[4]]
vig5 <- vig_r[[5]]
# Step 6: Merging each Specie's sub_subsets for 5 times
iris1 <- do.call("rbind", list(set1,ver1,vig1))
iris2 <- do.call("rbind", list(set2,ver2,vig2))
iris3 <- do.call("rbind", list(set3,ver3,vig3))
iris4 <- do.call("rbind", list(set4,ver4,vig4))
iris5 <- do.call("rbind", list(set5,ver5,vig5))
iris.5fold <- list(iris1, iris2, iris3, iris4, iris5)
# Step 7: Check each 5 dataset to see if the proportion is correct
summary(iris1$Species)
```

```
##     setosa versicolor  virginica
##         10         10         10
```

```
summary(iris2$Species)
```

```
##     setosa versicolor  virginica
##         10         10         10
```

```
summary(iris3$Species)
```

```
##     setosa versicolor  virginica
##         10         10         10
```

```
summary(iris4$Species)
```

```
##     setosa versicolor  virginica
##         10         10         10
```

```
summary(iris5$Species)
```

```
##     setosa versicolor  virginica
##         10         10         10
```

Distributions

R is endowed with a set of statistical tables. To obtain the density function, cumulative distribution function (CDF), quantile (inverse CDF) and pseudo-random numbers from a specific distribution, one only needs to prefix the distribution name given below by `d`, `p`, `q` and `r` respectively.

Distributions	R Names	Key Arguments
Uniform	`unif`	`min`, `max`
Normal	`norm`	`mean`, `sd`
χ^2	`chisq`	`df`, `ncp`
Student's t	`t`	`df`, `ncp`
F	`f`	`df1`, `df2`, `ncp`
Exponential	`exp`	`rate`
Gamma	`gamma`	`shape`, `scale`
Beta	`beta`	`shape1`, `shape2`, `ncp`
Logistic	`logis`	`location`, `scale`
Binomial	`binom`	`size`, `prob`
Poisson	`pois`	`lambda`

Geometric	`geom`	`prob`
Hypergeometric	`hyper`	`m , n , k`
Negative Binomial	`nbinom`	`size , prob`

Check from their plots.

The following two-sample t-test shows the usages of `qt , pt` and `rnorm` . Recall that a two-sample homoscedastic t-test statistic is

$$\hat{\sigma}^2 = \frac{(n_X-1)S_X^2+(n_Y-1)S_Y^2}{n_X+n_Y-2}, \quad T = \frac{\bar{X}-\bar{Y}}{\hat{\sigma}\sqrt{\frac{1}{n_X}+\frac{1}{n_Y}}} \overset{d}{\sim} t_{n_X+n_Y-2} \text{ under } H_0: \mu_X = \mu_Y.$$

Data Exploration and Manipulation

Data analysis in **R** starts with reading data in a data.frame object via `scan` and `read.table` as discussed before. Then one would explore the profiles of data via various descriptive statistics whose usages are also introduced in the previous sections. Calling the `summary` function with a data.frame input also provides appropriate summaries, *e.g.* means and quantiles for numeric variables (columns) and frequencies for factor variables.

This section explores more features that can be achieved through **R**.

Tables

Statistician often works with catergorical variables via tables. Even for continuous variables, segregating them into catergorical ones in a meaningful way might provide more insights. `table` function generates frequency tables for factor variables. Multi-way tables, marginal and proportional displays and independence test are explored in the following example. Recall that the Pearson's χ^2 independence test statistic on an r-by-c contingency table

$$\chi^2 = \sum_{i=1}^{r}\sum_{j=1}^{c} \frac{\left(n_{ij}-\frac{n_{i\cdot}n_{\cdot j}}{n_{\cdot\cdot}}\right)^2}{n_{i\cdot}n_{\cdot j}/n_{\cdot\cdot}} \overset{d}{\approx} \chi^2_{(r-1)(c-1)} \text{ under } H_0: p_{ij} = p_i p_j.$$

Plots

Compared to other statistical softwares in data analysis, **R** is good at graphic generation and manipulation. The plotting functions in **R** can be classified into the high-level ones and the low-level ones. The high-level functions create complete, new plots on the graphics device while the low-level functions only add extra information to the current plots.

`plot` is the most generic high-level plotting function in **R**. It will be compatble with most class of objects that the user input and produce appropriate graphics. For example, a numeric vector input results in a scatter plot with respect to its index and a factor vector results in a bar plot of its frequency table. Advanced class of

object like `lm` (fitted result by a linear model) can also be called in `plot`. Methods will be discussed in specific documents like `?plot.lm`.

Other plotting features include

- High-level plotting options: `type`, `main`, `sub`, `xlab`, `ylab`, `xlim`, `ylim`
- Low-level plotting functions
  - **Symbols:** `points`, `lines`, `text`, `abline`, `segments`, `arrows`, `rect`, `polygon`
  - **Decorations:** `title`, `legend`, `axis`
- Environmental graphic options (`?par`)
  - **Symbols and texts:** `pch`, `cex`, `col`, `font`
  - **Lines:** `lty`, `lwd`
  - **Axes:** `tck`, `tcl`, `xaxt`, `yaxt`
  - **Windows:** `mfcol`, `mfrow`, `mar`, `new`
- User interaction: `location`

I'll suggest the beginners learn from examples and grab when needed instead of going over such an overwhelming brochure. The following sections illustrate two basic senarios in data analysis. More high-level plotting functions will also be introduced.

Access the empirical distribution

Histogram: The `hist` function creates a histogram in **R**, where the `breaks` and `freq` options are frequently called. The `barplot` function could also realize the same result as illustrated in section **Tables**.

Kernel Density Curve: The `density` function estimates an empirical density of the data and gives a `density` object. One can call `plot` function with such an object as input and picture a density curve, which is anticipated to closely envelop its historgram. Note that the `bw` (bandwidth) and `kernel` options should be carefully considered in methodology.

Empirical CDF (ECDF): The `ecdf` function generates an empirical CDF (`"ecdf"`) object that can be called by the `plot` function, which results in a step function graphic for the empirical cumulative distribution function. Creating a graph containing multiple CDFs or ECDFs visually displays the good-of-fitness or discrepancy among them. The statistically quantified Kolmogorov-Smirnov test with statistic

$$D_n = \sup_{x \in \mathbb{R}} |F_n(x) - F(x)|$$

realized by the `ks.test` function is based on it.

Q-Q Plots: "Q-Q" stands for the sample quantiles versus that of a given distribution or another sample. `qqnorm`, `qqline` and `qqplot` together is the set of functions in realizing it. **R** documents also illustrate how to create Q-Q plots against distributions other than Normal.

Box-and-Whisker Plots: With `boxplot` function in **R**, we can describe the data with box associated with certain quantiles and the whiskers for extremes. The box in the middle indicates "hinges" (nearly quartiles, see `?boxplot.stats`) and median. The lines ("whiskers") show the largest/smallest observation that falls within a distance of 1.5 times the box size from the nearest hinge. If any observations fall farther away, the additional points are considered "extreme" values and are shown separately.

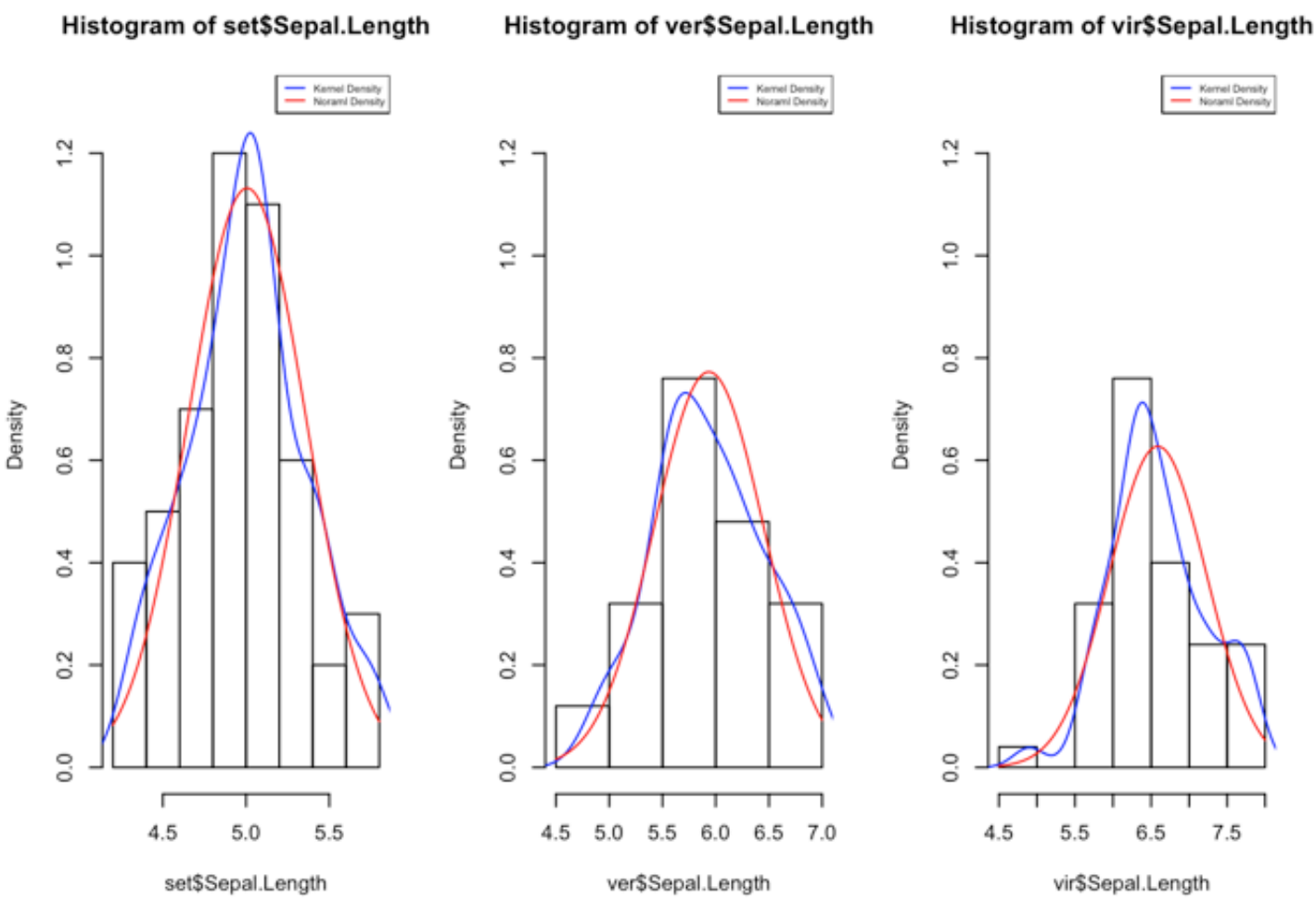
Exercise 11. *(15 pt)* Reproduce the code that generates the following plot about `Sepal.Length` in `iris`.

Hint. 1) Most decorations are based on defaults in `hist` with `ylim = c(0, 1.3)`; 2) let `h` be the object resulted by `hist` and set `xlim = range(h$breaks)`; 3) set `cex = 0.5` in `legend`; 4) use `curve` function in plotting the normal density with specified parameters.

```
set = subset(iris,Species=='setosa')
ver = subset(iris,Species=='versicolor')
vir = subset(iris,Species=='virginica')
par(mfrow=c(1,3))
hist(set$Sepal.Length,freq = FALSE,ylim=c(0,1.3))
lines(density(set$Sepal.Length),col='blue')
curve( dnorm(x, mean=mean(set$Sepal.Length),sd=sd(set$Sepal.Length)), add=T, col="red
")
legend("topright",legend=c("Kernel Density","Noraml Density"),col=c('blue','red'),lty
=1,cex=0.5)

hist(ver$Sepal.Length,freq = FALSE,ylim=c(0,1.3))
lines(density(ver$Sepal.Length),col='blue')
curve( dnorm(x, mean=mean(ver$Sepal.Length),sd=sd(ver$Sepal.Length)), add=T, col="red
")
legend("topright",legend=c("Kernel Density","Noraml Density"),col=c('blue','red'),lty
=1,cex=0.5)

hist(vir$Sepal.Length,freq = FALSE,ylim=c(0,1.3))
lines(density(vir$Sepal.Length),col='blue')
curve( dnorm(x, mean=mean(vir$Sepal.Length),sd=sd(vir$Sepal.Length)), add=T, col="red
")
legend("topright",legend=c("Kernel Density","Noraml Density"),col=c('blue','red'),lty
=1,cex=0.5)
```



Demostrate correlation

`plot(x, y)` directly creates a scatter plot between the vector `x` and `y`. For a data.frame input `X`, `plot(X)` would conduct pairwise scatter plots among its columns. A fitted regression line can also be added to an existing scatter plot via the `abline` function. And the function `coplot` is a power function in creating conditioning plots given a variable segregated into different levels.

*Create advanced plots

If you are a JAVA programmer, then you might anticipate a plotting toolbox to establish graphs layer-by-layer interactively. The `ggplot2` package endows **R** with more advanced and powerful visualization techniques like this. Explore more in the online package manual (https://cran.r-project.org/web/packages/ggplot2/ggplot2.pdf). The following example solves **Exercise 11** without segregating with respect to `Species` in `iris`.

*Data Manipulation

If you are more familiar with the SQL language in manimulating data, then the `dplyr` package in **R** is a powerful toolkit in providing functions similar to the SQL operations (*e.g.* `select`, `filter`, `arrange`, `mutate`, `inner_join`, `group_by`, `summarise` and the pipe operator `%>%`). The following example realizes

the conditional selection technique in the previous exercises in a much cleaner way. Explore more in the online package mannual (https://cran.r-project.org/web/packages/dplyr/dplyr.pdf).

Bibliography

Dalgaard, Peter. 2008. *Introductory Statistics with R: Statistics and Computing*. Springer.

W. N. Venables, D. M. Smith, and the R Core Team. 2017. *An Introduction to R: Notes on R: A Programming Environment for Data Analysis and Graphics* (version 3.4.3).

Maximum Likelihood

Detail: $\log L\,(\theta|x) = \sum_{i=1}^{n} logp(x_i|\theta)$

Gaussian probability density function:

$$P(X = x|\mu, \sigma^2) = \sum_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\,(-\frac{1}{2\sigma^2}(x_i - \mu)^2)$$

Step 1: Find the log-likelihood function of the Gaussian probability density function

Log-likelihood function:

$$\sum_{i=1}^{n} \log(x|\mu, \sigma^2) = \log\,(\sum_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x_i - \mu)^2\right))$$

$$= \log\,[\sum_{i=1}^{n}(2\pi\sigma^2)^{-0.5}] + \log\,[\exp\left(-\tfrac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2\right)]$$

$$= \log\,[(2\pi\sigma^2)^{-0.5n}] + (\sum_{i=1}^{n} -\frac{1}{2\sigma^2}(x_i - \mu)^2)$$

$$= -0.5n * \log(2\pi\sigma^2) + (\sum_{i=1}^{n} -\frac{1}{2\sigma^2}(x_i - \mu)^2)$$

Step 2: Compute the first derivative of the log-likelihood function

For parameter μ:

$$\frac{\partial(logf)}{\partial\mu} = (-0.5n)'(log2\pi\sigma^2) + (log2\pi\sigma^2)'(-0.5n) + \left(-\frac{1}{2\sigma^2}\right)'\left(\sum_{i=1}^{n}(x_i - \mu)^2\right) + \left(\sum_{i=1}^{n}(x_i - \mu)^2\right)'\left(-\frac{1}{2\sigma^2}\right)$$

$$= 0 + 0 + \left(-\tfrac{1}{2\sigma^2}\right) * 2 * \sum_{i=1}^{n}(\mu - x_i)$$

$= \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i - \mu)$

For parameter σ^2

Set a = σ^2

$$\frac{\partial(logf)}{\partial a} = (-0.5n)'(log2\pi a) + (log2\pi a)'(-0.5n) + \left(-\frac{1}{2a}\right)'\left(\sum_{i=1}^{n}(x_i - \mu)^2\right) + \left(\sum_{i=1}^{n}(x_i - \mu)^2\right)'\left(-\frac{1}{2a}\right)$$

$= \frac{-n}{2a} + \frac{1}{4a^2}\sum_{i=1}^{n}(x_i - \mu)^2$

$= \frac{1}{2a}(-n + \frac{1}{a}\sum_{i=1}^{n}(x_i - \mu)^2)$

Because a = σ^2, then we have

$\frac{\partial(logf)}{\partial a} = \frac{1}{2\sigma^2}(-n + \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2)$

Step 3: solve the log-likelihood functions equals to 0 to find the MLE.

Solve $\frac{\partial(logf)}{\partial \mu} = 0$

$\frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i - \mu) = 0$

$$\sum_{i=1}^{n}(x_i - \mu) = 0$$

$$\sum_{i=1}^{n}\mu = \sum_{i=1}^{n}x_i$$

$$n * \mu = \sum_{i=1}^{n} x_i$$

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Solve $\frac{1}{2\sigma^2}(-n + \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2) = 0$

$$\frac{1}{2\sigma^2}(-n + \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2) = 0$$

$$(-n + \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2 = 0$$

$$\frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2 = n$$

$$\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 = \sigma^2$$

Therefore,

$$\mu_{MLE} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\sigma^2{}_{MLE} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$