STOR 767 Spring 2019 Hw1: Computational Part

Due on 01/23/2019 in Class

YOUR NAME

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.

- Homework 1 includes **Theoretical Part** (50%) and **Computational Part** (50%).
- Download the whole folder for this homework. Open the **RMarkdown** file "Hw1_Intro_R.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 example code chunks, the code outputs and the plots would display in the created file.
- Exercises in-between are to be accomplished. Any packages and functions can be employed if no additional requirements are announced.
- For homework submission and grading, edit "Hw1_Intro_R_sb.Rmd" and create a PDF file to print and submit in class. Codes and key results should be displayed.
- For more information about the **RStudio**, refer to the section **Getting Started**; about the **RMarkdown**, refer to the online tutorial[1] and the online manual of `knitr`[2] by Yihui Xie from **RStudio, Inc.**

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[3] of German Rodriguez from Princeton. To download and install **R**, click into the CRAN (Comprehensive R Archive Network) Mirrors 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 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), (W. N. Venables and R Core Team 2018) and the tutorial 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.

[1] http://rmarkdown.rstudio.com/lesson-1.html.
[2] *`knitr`: Elegant, flexible, and fast dynamic report generation with R* https://yihui.name/knitr/.
[3] *Introducing R* http://data.princeton.edu/R.

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.

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 1. *(5 pt)* **Hadamard matrix** is a useful construction for two-level orthogonal design. It's defined recursively by

$$\mathbf{H}_1 = \begin{pmatrix} 1 \end{pmatrix} \in \mathbb{R}^{1\times 1}, \quad \mathbf{H}_{2^k} = \begin{pmatrix} \mathbf{H}_{2^{k-1}} & \mathbf{H}_{2^{k-1}} \\ \mathbf{H}_{2^{k-1}} & -\mathbf{H}_{2^{k-1}} \end{pmatrix} \in \mathbb{R}^{2^k \times 2^k}. \quad (k \in \mathbb{N})$$

Create $\mathbf{H}_{2^4}$ in **R**.

```
YOUR CODE HERE
```

Indexing

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

```
x <- 1:4

# specify the vector of indices
x[c(1,4)]
```

```r
# specify a logical vector of identical length
x[c(TRUE, FALSE, FALSE, TRUE)]

# specify the indices to discard
x[-c(2,3)]
```

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

```r
x <- 1:4
x >= 3    # componentwise comparison resulting in a logical vector
x[x >= 3]
```

Matrix indexing follows the same manner.

```r
x <- matrix(1:12, 3, 4)
x[c(1,3),-c(1,4)]
x[c(TRUE,FALSE,TRUE),]
```

Exercise 2. *(5 pt)* It has been shown that a LASSO estimate for the location of X is of the following thresholding form

$$\hat{\mu}_{\text{LASSO}} = \underset{\mu\in\mathbb{R}}{\operatorname{argmin}} \frac{1}{2}(X-\mu)^2 + \lambda|\mu| = \begin{cases} X+\lambda, & X \leqslant -\lambda \\ 0, & -\lambda < X \leqslant \lambda \\ X-\lambda, & X > \lambda \end{cases}$$

Now let $\lambda = 1$ and consider 100 *i.i.d.* sample $\{X_i\}_{i=1}^n$ drawn from $\mathcal{N}(0,1)$. Return the vector of the LASSO estimates for their individual locations in **R**.

```r
x <- rnorm(100)
YOUR CODE HERE
```

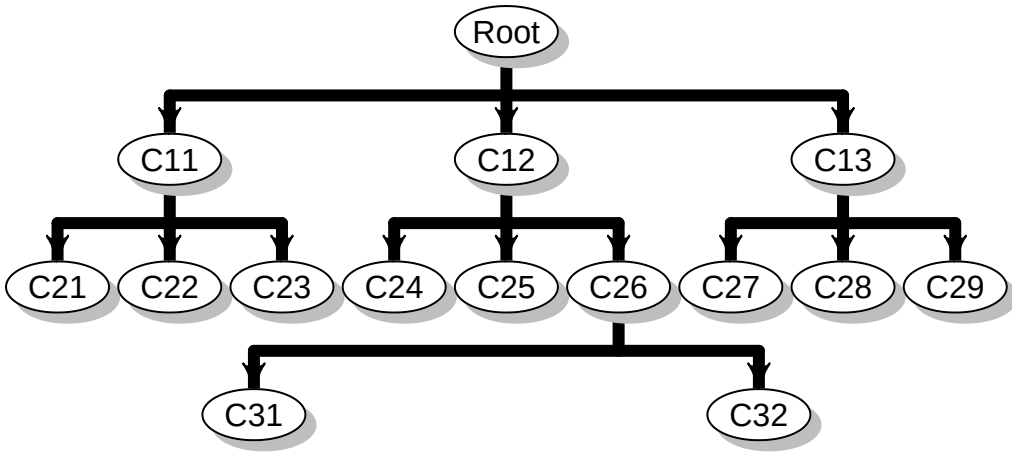
List

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,

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

Punchline. Only ONE index instead of a vector of indices can be placed into the double bracket! Explore in the following example to see the difference as compared to the single bracket indexing.




```
C11 <- list( C21 = "C21",
             C22 = "C22",
             C23 = "C23")
C26 <- list( C31 = "C31",
             C32 = "C32")
C13 <- list( C27 = "C27",
             C28 = "C28",
             C29 = "C29")
C12 <- list( C24 = "C24",
             C25 = "C25",
             C26 = C26)
Root <- list( C11 = C11,
              C12 = C12,
              C13 = C13)

# subtree rooted at C12
Root[[2]]
Root$C12

# subtree (leaf) rooted at C24
Root[[c(2,1)]]
Root$C12$C24

# subtree rooted at C26
Root[[c(2,3)]]
Root$C12$C26

# subtree (leaf) rooted at C31
Root[[c(2,3,1)]]
Root$C12$C26$C31
```

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

```
scores <- matrix(c(95, 97, 90, 80, 75, 84), 3, 2)
scores <- data.frame(scores)
colnames(scores) <- c("score_A", "score_B")
id <- c("001", "002", "003")
students1 <- cbind(id, scores)
students2 <- data.frame( list( id      = c("001", "002", "003"),
                               score_A = c(95, 97, 90),
                               score_B = c(80, 75, 84))
                         )
```

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 difference

# 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
students3_copy <- students3    # work on a copy in case of direct modification of students3
levels(students3_copy$id)[3] <- "002"
levels(students3_copy$id)
students3_copy
```

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 3. *(5 pt)* Table 1 presents a mixed 2-level and 3-level orthogonal design from (Wu and Hamada 2011). The first four rows in 2-level factors A, B and C, as a 2^{3-1} design, have been repeated for the next eight 4-row groups. Groups are embedded into a 3^{3-1} design in 3-level factors D, E and F. In particular, column C = column A $\times$ column B, column F = column D + column E (mod 3) by encoding $(-1, 0, 1)$ in $(1, 2, 0)$. Create such design matrix in **R** without reading from Table 1 directly.

```
YOUR CODE HERE
```

Table 1 $2^{3-1} \times 3^{3-1}$ Orthogonal Array

	2-Level Factors			3-Level Factors		
	A	**B**	**C**	**D**	**E**	**F**
1	-1	-1	1	-1	-1	-1
2	1	-1	-1	-1	-1	-1
3	-1	1	-1	-1	-1	-1
4	1	1	1	-1	-1	-1
5	-1	-1	1	0	-1	0
6	1	-1	-1	0	-1	0
7	-1	1	-1	0	-1	0
8	1	1	1	0	-1	0
9	-1	-1	1	1	-1	1
10	1	-1	-1	1	-1	1
11	-1	1	-1	1	-1	1
12	1	1	1	1	-1	1
13	-1	-1	1	-1	0	0
14	1	-1	-1	-1	0	0
15	-1	1	-1	-1	0	0
16	1	1	1	-1	0	0
17	-1	-1	1	0	0	1
18	1	-1	-1	0	0	1
19	-1	1	-1	0	0	1
20	1	1	1	0	0	1
21	-1	-1	1	1	0	-1
22	1	-1	-1	1	0	-1
23	-1	1	-1	1	0	-1
24	1	1	1	1	0	-1
25	-1	-1	1	-1	1	1
26	1	-1	-1	-1	1	1
27	-1	1	-1	-1	1	1
28	1	1	1	-1	1	1
29	-1	-1	1	0	1	-1
30	1	-1	-1	0	1	-1
31	-1	1	-1	0	1	-1
32	1	1	1	0	1	-1
33	-1	-1	1	1	1	0
34	1	-1	-1	1	1	0
35	-1	1	-1	1	1	0
36	1	1	1	1	1	0

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 4. *(5 pt)* Thickness data from a paint experiment based on Table 1 design in (Wu and Hamada 2011) are collected as below. Compute the sum of squares for all factors (main effects) from scratch, *i.e.* without resorting to any ANOVA-type **R** functions. Compare them with outputs produced by `aov`.

```
y <- scan(text = "0.755 0.550 0.550 0.600 0.900 0.875 1.000 1.000 1.400 1.225 1.225 1.475
0.600 0.600 0.625 0.500 0.925 1.025 0.875 0.850 1.200 1.250 1.150 1.150 0.500 0.550 0.575
0.600 0.900 1.025 0.850 0.975 1.100 1.200 1.150 1.300")

YOUR CODE HERE
```

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 difference
  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

```
mymax.if <- function(x, y)
{
  # The mymax function returns the larger one among x and y
  if (x > y)
  {
    message("The first argument is larger!")
    return(x)
```

```
  }
  else if (x < y)
  {
    message("The second argument is larger!")
    return(y)
  }
  else
  {
    message("Equal!")
    return(x)
  }
}
mymax.if(3, 4)
mymax.if(4, 3)
mymax.if(3, 3)
```

Loops

```
factorial <- function(n)
{
  # It computes the factorial of the natural number n
  if(n == 0) return(1) else if(n < 0 | n != as.integer(n)) stop("Please input a natural number!")
  n.fac <- 1

  # for loops
  for(i in 1:n)
    n.fac <- i * n.fac

  # while loops
  ## i <- 1
  ## while(i <= n)
  ## {
  ##   n.fac <- i * n.fac
  ##   i <- i + 1
  ## }

  # repeat loops
  ## i <- 1
  ## repeat
  ## {
  ##   n.fac <- i * n.fac
  ##   i <- i + 1
  ##   if (i > n) break    # attention to the usage of break
  ## }

  return(n.fac)
}

double.factorial <- function(n)
{
  # It computes the double factorial of the natural number n
  if(n == 0) return(1) else if(n < 0 | n != as.integer(n)) stop("Please input a natural number!")
```

```
  n.doubfac <- 1
  for(i in 1:n)
    if(i %% 2 == n %% 2)
      n.doubfac <- i * n.doubfac
    else next    # attention to the usage of next
  return(n.doubfac)
}

factorial(6)
prod(1:6)
double.factorial(6)
double.factorial(5)
factorial(0)
## factorial(6.5)
## factorial(-6)
```

Exercise 5. *(10 pt)* Write a function `optim_gd(par, fn, gr, gr_lips, maxit = 10000, tol = 1e-5)` to find the minimizer of a smooth convex function using gradient descent.[4]

- `par`: initial values for the parameters to be optimized over.
- `fn`: objective function to be minimized f on domain $\mathcal{X}$.
- `gr`: gradient of objective function ∇f.
- `gr_lips`: Lipschitz gradient constant L_f, *i.e.*

$$\|\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y})\|_2 \leqslant L_f \|\boldsymbol{x} - \boldsymbol{y}\|_2. \quad (\forall \boldsymbol{x}, \boldsymbol{y} \in \mathcal{X})$$

- `maxit`: maximal number of iterations.
- `tol`: convergence tolerance parameter $\epsilon > 0$.

Iterations are performed by

$$\boldsymbol{x}^{k+1} := \boldsymbol{x}^k - \frac{1}{L_f} \nabla f(\boldsymbol{x}^k)$$

with stopping criterion

$$\frac{\|\nabla f(\boldsymbol{x}^k)\|_2}{\max\{1, \|\nabla f(\boldsymbol{x}^0)\|_2\}} \leqslant \epsilon.$$

Return a list with `par =` minimizer, `value =` optimal objective value, and `counts =` number of iterations performed. Apply it to the bivariate function[5]

$$f(x_1, x_2) = \log\left(1 + e^{-x_1 + x_2}\right) + \log\left(1 + e^{x_1}\right) + \log\left(1 + e^{-x_1 - x_2}\right)$$

with $L_f = \frac{5}{4}$ and initial value $(0, 0)$. Compare it with the built-in optimization function `optim` with `method = "BFGS"`.

```
YOUR CODE HERE
```

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.

[4] STOR 893 Fall 2018 lecture note http://quoctd.web.unc.edu/files/2018/10/lecture4-selected-cvx-methods.pdf.

[5] Negative log-likelihood of Logistic regression of the data $\{(-1, 1), (0, 0), (1, 1)\}$. Try performing `glm` to see whether the outputs coincide.

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 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.

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.

```
plot(dnorm, xlim = c(-5, 5))   # bell curve of Normal density
plot(plogis, xlim = c(-5, 5))  # Logistic/Sigmoid function (CDF of Logistic distribution)
```

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

t-test statistic is

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

```
twosam <- function(x, y, alpha = 0.05)
{
  # It conducts a two-sample homoscedastic t-test on x and y
  n.x       <- length(x); n.y    <- length(y)
  mean.x    <- mean(x);   mean.y <- mean(y)
  var.x     <- var(x);    var.y  <- var(y)
  mean.diff <- mean.x - mean.y
  df        <- n.x + n.y - 2
  sigma     <- ((n.x - 1) * var.x + (n.y - 1) * var.y) / df
  var.diff  <- (1/n.x + 1/n.y) * sigma
  t         <- mean.diff / sqrt(var.diff)
  t.alpha   <- qt(1 - alpha/2, df)
  output    <- list(t       = t,
                    df      = df,
                    p.value = 2 * pt(-abs(t), df),
                    confint = c(lower = mean.diff - sqrt(var.diff) * t.alpha,
                                upper = mean.diff + sqrt(var.diff) * t.alpha),
                    mu      = c(mu.x = mean.x, mu.y = mean.y),
                    sigma   = sigma)
  return(output)
}
x1 <- rnorm(40, 0, 1)
x2 <- rnorm(50, 0, 1)
x3 <- rnorm(50, 1, 1)
twosam(x1, x2)
t.test(x1, x2, var.equal = TRUE)
twosam(x1, x3)
t.test(x1, x3, var.equal = TRUE)
```

Exercise 6. *(10 pt)* Create the ANOVA table based on **Exercise 4** from scratch. Sum of squares, degrees of freedom, F-values, p-values and indicators of significance are to be reported.

```
YOUR CODE HERE
```

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

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

```
# Discretize the numeric variables
iris0 <- transform(iris,
                   Sepal.Length = cut(Sepal.Length, 4:8),
                   Sepal.Width  = cut(Sepal.Width,  1:5),
                   Petal.Length = cut(Petal.Length, 0:7),
                   Petal.Width  = cut(Petal.Width,  0:3))

attach(iris0)   # attach function makes columns assessible as usual variables
table(Sepal.Width)
iris.table0 <- table(Petal.Width)
iris.table0
iris.table1 <- table(Sepal.Width, Petal.Width)
iris.table1
iris.table2 <- table(Sepal.Width, Petal.Width, Species)
iris.table2

# flat table in a sligtly different display
ftable(Sepal.Width, Petal.Width)
ftable(Species, Sepal.Width, Petal.Width)

# data.frame display
data.frame(iris.table1)
data.frame(iris.table2)

# marginal display
margin.table(iris.table1, 1)
margin.table(iris.table1, 2)

# proportional display relative to the (margin) total
prop.table(iris.table1)
prop.table(iris.table1, 1)
prop.table(iris.table1, 2)
prop.table(iris.table2, 1)
prop.table(iris.table2, c(1,3))

# conduct chi-square independence test
chisq.test(iris.table1)   # warning message due to some frequency entry <= 5

# plots
pie(iris.table0, main="Petal.Width")
barplot(t(iris.table1), beside = TRUE,
        xlab = "Sepal.Width", ylab = "Petal.Width", legend.text = colnames(iris.table1))

detach(iris0)   # paired with attach function
```

*Generate formatted tables

You might have found it unaesthetic to display tables from **R** console outputs and would like to display their LaTex-formatted version handfully. The `kable` function from `knitr` and `kableExtra` packages automatically turn an `data.frame` object into formatted table. Explore more features in a wonderful online tutorial[6] by Hao Zhu. The following example create a formatted version of the long display (`data.frame` display) of `iris.table2` above.

```
kable(data.frame(iris.table2), format = "latex", booktabs = T, align = "l",
      caption = "Sepal-Petal Frequency Table Grouped by Species") %>%
  kable_styling(position = "center", full_width = T,
                latex_options = c("hold_position", "striped")) %>%
  row_spec(0, bold = T) %>%
  group_rows("setosa", 1, 12) %>%
  group_rows("versicolor", 13, 24) %>%
  group_rows("virginica", 25, 36)
```

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**.

[6] *Create Awesome HTML Table with `knitr::kable` and `kableExtra`* https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome_table_in_html.html.

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

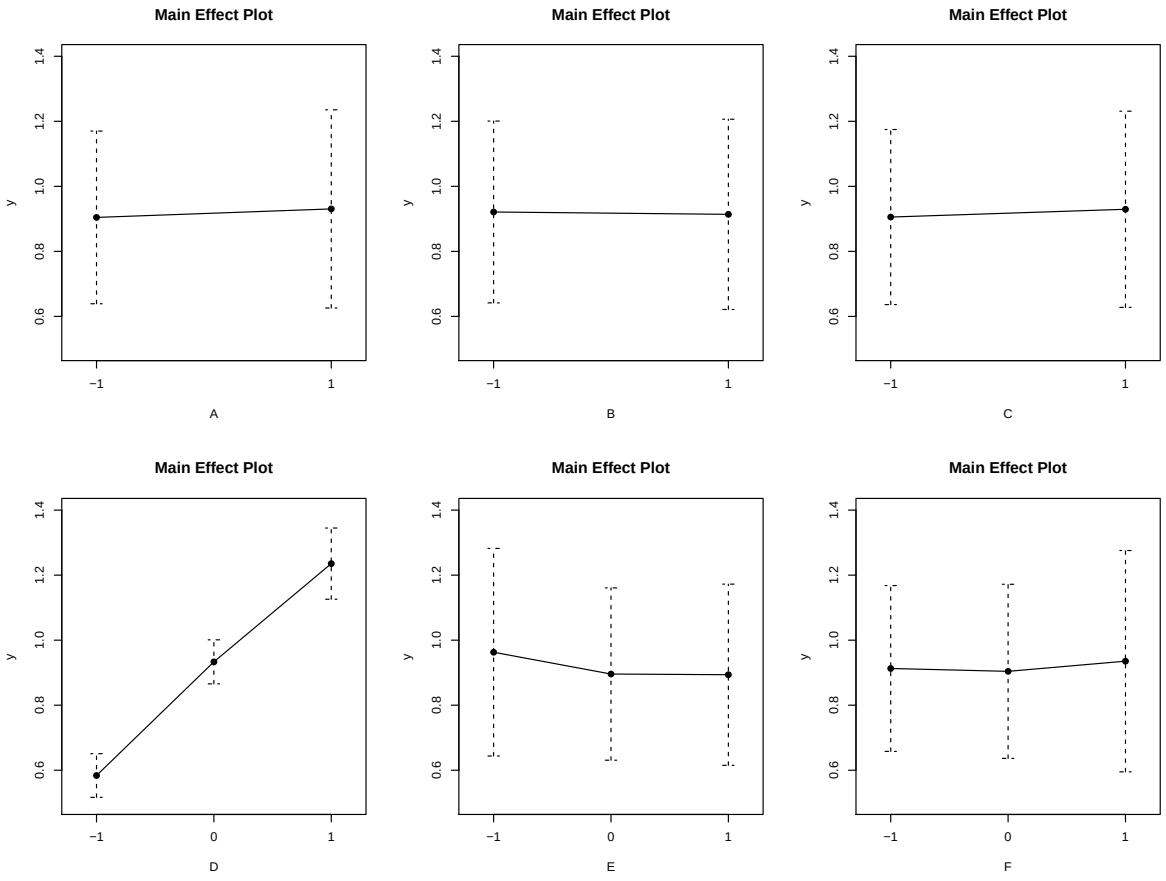
$$D_n = \sup_{x \in \mathbb{R}} |F_n(x) - F(x)|$$

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 7. *(10 pt)* Reproduce the code that generates the following plot based on **Exercise 4**.



```
YOUR CODE HERE
```

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.

```
plot(iris[, -5])

attach(iris)  # attach function makes columns assessible as usual variables
lm.mod <- lm(Petal.Length ~ Sepal.Length)   # fit a linear model as an lm object
plot(Sepal.Length, Petal.Length)
## plot(Petal.Length ~ Sepal.Length)
abline(lm.mod)

coplot(Petal.Length ~ Sepal.Length | Species)
detach(iris)  # paired with attach function
```

*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 ggplot2 homepage.[7] The following example creates the histograms and density curves (normal and nonparametric estimates) for the distributions of `Sepal.Length` in `iris` dataset stratified to `Species` in `iris`.

```
ggplot(iris, aes(Sepal.Length)) +
  geom_histogram( aes(y = ..density..),
                  bins = 8, color = "black", fill = "white") +
  geom_density(aes(color = "blue")) +
  stat_function( aes(color = "red"),
                 fun = dnorm, args = list(mean = mu, sd = sigma)) +
  labs(title = "Histogram of Sepal_Length") +
  scale_color_identity( name   = "Density Estimate",
                        guide  = "legend",
                        labels = c("Kernel", "Normal")) +
  theme_bw()
```

*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 techniques in a much cleaner way. Explore more in `dplyr` homepage.[8]

```
# Creating summary variables across subj subgroups
students3_group <- group_by(students3, subj)
mutate(students3_group,
```

[7] https://ggplot2.tidyverse.org.
[8] https://dplyr.tidyverse.org.

```
      score.mean = mean(score),
      score.sd   = sd(score),
      score.min  = min(score),
      score.med  = median(score),
      score.max  = max(score))

# Using the pipe operator makes it more compact
group_by(students3, subj) %>%
  mutate(score.mean = mean(score),
         score.sd   = sd(score),
         score.min  = min(score),
         score.med  = median(score),
         score.max  = max(score))
```

Bibliography

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

W. N. Venables, D. M. Smith, and the R Core Team. 2018. *An Introduction to R: Notes on R: A Programming Environment for Data Analysis and Graphics* (version 3.5.2). https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf.

Wu, C.F. Jeff, and Michael S. Hamada. 2011. *Experiments: Planning, Analysis, and Optimization*. Vol. 552. John Wiley & Sons.