Computer Assignment 1 - Introduction to R

Machine Learning, Spring 2020

Rui Li

Remark. The purpose of this homework is to introduce you to the basics of **R**: the main software we will use throughout this course. This is a long homework assignment aimed to be a sort of *R Bootcamp*. While it is lengthy, much of this assignment consists of explanations regarding basic things in R.

Instruction. Open the **RMarkdown** document with suffix ".rmd" via **RStudio**. Click `Knit` to create a PDF document (remember to install the necessary packages as described in CA0). The "knit" option can be changed to pdf *or* html (or even Word). When submitting the HW, you **will need to knit to a pdf document and print the output for submission in class**. The html file in the folder has clickable links for the various references below on Latex etc. By removing the `results='hide'` and `fig.keep='none'` options in the code chunks, the code outputs and the plots will display in the created file. For more information about **RStudio**, refer to the section **Getting Started**; for more information about **RMarkdown**, refer to the online tutorial and the online manual of knitr by Yihui Xie from **RStudio, Inc.**

Important caveat: In the past some students have had issues when knitting the file to pdf. If your work does not show up, knit to html or doc (this seems to usually work fine) and print the output. At the end of the day all we need to grade your HW is a paper copy of RMarkdown's output. Please note that this does NOT mean it is alright to simply turn in a `.Rmd` file.

Latex: If the file does not compile properly in the initial go around you might need to install Latex onto your system. See Latex installation for details. You might then need to re-install RStudio. Latex is a fantastic framework that everyone in the computational world uses to write technical documents. See Tex exchange or Medium article for more details as to why you should use Latex. See this guide for a beginners introduction.

diagram package: For displaying some of the pictures in the file we used an R package called diagram. You will need to install this before the file knits properly.

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.

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. This reflects how we as humans understand data; data can take many forms and look differently depending on what we are observing. Think of a list of test scores. These data are all numeric (real numbers between 0 and 100), and are only one-dimensional. In contrast, a traditional data set with multiple rows and columns is multidimensional, and also is not usually limited to just numerical observations. To understand a data set in R, one should first understand which class of object he/she/they has on hand, and then figure out the applicable operations on it.

In a hierarchical manner, the more advanced classes consist of ingredients from more fundamental classes. Vectors, matrices, lists, and data.frames are the most commonly used fundamental classes in data analysis. So, these next few sections will explain these data types, as well as provide some motivation for why we have them in the first place.

Vectors and Matrices

A vector is a collection of "data" that share the same type (numeric, character, logic or NULL). A 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 concatenation 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 is `:`, which similar to `seq` and also creates the consecutive sequence with step sizes by 1 or -1. Try running `1:4` and `4:1`.

```
x <- seq(1,4,1); y <- seq(4,1,-1)
x0 <- c(1:4); y0 <- c(4:1)
x;y

## [1] 1 2 3 4

## [1] 4 3 2 1

x0;y0

## [1] 1 2 3 4

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

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

Exercise 1 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 = (“Hello”," “,”World“,”!“,”Hello World!");

Note: The quotation marks and sometimes the exclamations marks are rendered a little funky in the pdf/html. Just go with it.

Hint. For x4, take this opportunity to experiment with the `paste` function.

x5 = (TRUE, TRUE, NA, FALSE);

Remark. Check `?NA` and `class(NA)` to learn more about the missing value object `NA`. This is not relevant for x5.

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

```
x1 = c(2,.5,4,2)
x2 = c(x1, rep(1,4))
x3 = c(1:-2);

x4 = c("Hello"," ","World","!",paste("Hello"," ","World","!"))

x5 = c(TRUE, TRUE, NA, FALSE)
x6 = c(rep(1:2,2),rep(1,2),rep(2,2))
x1; x2; x3; x4; x5; x6

## [1] 2.0 0.5 4.0 2.0

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

## [1]  1  0 -1 -2

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

## [5] "Hello   World !"

## [1]  TRUE  TRUE    NA FALSE

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

Create a Matrix

A 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 as follows: `x <- matrix(1:4, 2, 2, byrow = TRUE)`.

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

Exercise 2. Using the `matrix` and `rbind` functions, create

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

To be more precise, first define a set of four vectors corresponding to the rows of the above matrix and then use rbind to make a corresponding matrix. Note: you will need to play around with the `deparse.level` option in `rbind` to get the matrix as above. Check out the manual page `?rbind` for more information.

```
a = matrix(1:4,1,4,byrow = TRUE)
b = matrix(1:-2,1,4,byrow = TRUE)
c = matrix(c(2,.5,4,2),1,4,byrow = TRUE)
d = matrix(rep(1,4),1,4,byrow = TRUE)
X = rbind(a,b,c,d)
X

##      [,1] [,2] [,3] [,4]
## [1,]    1  2.0    3    4
## [2,]    1  0.0   -1   -2
## [3,]    2  0.5    4    2
## [4,]    1  1.0    1    1
```

Indexing

There are a number of ways to extract specific components of a vector.

```
# First, we'll make a vector (1,2,3,4):
x <- 1:4

# Here, we grab the values at the first and fourth indices
# (which happen to be 1 and 4 in this case):
x[c(1,4)]

# We can also grab the first and fourth values this way:
x[c(TRUE, FALSE, FALSE, TRUE)]

# Or we can grab all values NOT at the second and third indices:
x[-c(2,3)]
```

Another approach uses conditional statements, which leads to the so called "conditional selection" technique as follows.

```
# Again, we'll define a vector:
x <- 1:4
# Observe this componentwise comparison resulting in a logical vector:
x >= 3
# Now observe what happens when we use this "logical vector" to index our vec
tor:
x[x >= 3]
# We can use & ("and") or | ("or") to get more sophistocated logical statemen
ts.
x[x >=1 & x <=3]
```

Matrix indexing follows similarly to how we index vectors. Indeed,

```
# We'll first make a 3x4 matrix with values rangeing from 1 to 12:
x <- matrix(1:12, 3, 4)
# This statement returns values that are in the first and third rows,
# but NOT in the first and fourth columns:
x[c(1,3),-c(1,4)]
# This statement returns values in the first and third rows,
# and any column:
x[c(TRUE,FALSE,TRUE),]
```

Exercise 3. Consider the matrix X from Exercise 2.

- Make a new vector y1 consisting of all the elements of X which are negative (strictly less than zero). Here you are expected to use a logical statement like the ones we saw in **Exercise 2**.

```
y1 <- X[X<0]
y1

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

- Make a new vector y2 consisting of all the elements of X which are at strictly positive but less than 2. Again, you should be using a logical statement.

```
y2 <- X[X>0 & X<2]
y2

## [1] 1.0 1.0 1.0 0.5 1.0 1.0 1.0
```

Lists

A list is a more flexible container of “data” that permits inhomogeneous types. That is, unlike vectors, the values in a list can vary between numeric and non-numeric types. This is 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,

```
# We'll make a list with many different types and formats.
# Text to the left of = specifies the component name:
x <- list( num   = 1:4,                # "num =" specifies the name of the fi
rst component
           chac  = "hello world!",
           logic = c(TRUE,FALSE),
           nu    = NULL,
           mat   = matrix(4:1, 2, 2) )
y <- list( 1234,
           "world" )
# We can still use the c() function to combine two lists into one list:
c(x, y)
```

To extract the components in a list, one should use double bracket `[[ ]]` instead of a single bracket. If one has 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 convenient alternative is the command `x$logic`.

Data Frames

A data.frame is a container that inherits key attributes from lists and matrices, which allows it to hold types of data that cannot be held in either of these original containers. Data.frames are more flexible in nature, and for this reason they are the most common container used in *R* programming. They permit 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). With the lists that we just learned, we can have inhomogeneous types, but we must be at most one-dimensional. With matrices, we can be multidimensional, but our types must be homogeneous. A data.frame is specifically the type that allows for multidimensional, inhomogeneous data.

Let's start with a motivating example. The following creates a score table of 3 students, where the first row contains character vectors and the last two rows contain numeric data:

```
students <- data.frame( id      = c("001", "002", "003"), # ids are character
s
                        score_A = c(95, 97, 90),          # scores are numeri
css
                        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 4. Applying the conditional selection technique (see the section "indexing" and do not use the *subset* function), extract the record of student 003 i.e their id number, and their scores in the two tests.

```
student_003 = students[students$id=="003",]
student_003

##    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.

```
# First, we create the matrix:
scores <- matrix(c(95, 97, 90, 80, 75, 84), 3, 2)
# Then, we convert the matrix into a data.frame:
scores <- data.frame(scores)
# Easy!!
# Now, let's name the columns
colnames(scores) <- c("score_A", "score_B")
# and add another column:
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))
                         )
```

Exercise 5. Create a data.frame object to display the calendar for Jan 2018 as follows. Use what we have learned so far about creating Vectors, Lists, and Matrices, then convert what you have created into a data.frame.

```
## 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
days <- matrix(0:31,5,7, byrow = TRUE)

## Warning in matrix(0:31, 5, 7, byrow = TRUE): data length [32] is not a sub
-
## multiple or multiple of the number of rows [5]

days[days==0 | days==1 | days==2] = ""
days[1,2] = "NY"; days[1,3] = "2"
days[3,2] = "MLK"

Jan2018 <- data.frame(days)
colnames(Jan2018) <- c("Sun", "Mon", "Tue", "Wed", "Thu", "Fri","Sat")

print(Jan2018,row.names = FALSE)

##  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
```

Ignore the ## symbols this was just so the above acts like a comment in R.

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

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.

Challenge Problem (not graded) `iris` is a built-in data set in **R**. Check `?iris` for more information. This data set has data on 50 flowers each from 3 species of Iris (setosa, versicolor, and virginica). Randomly divide `iris` into five subsets `iris1` to `iris5` (without replacement), thus each subset has 30 rows of the iris data and further stratified to `iris$Species` (namely every subset should have 10 rows from each of the 3 species). **Hint**: One solution to this problem involves first seperating the data by species type, then using the `sample` function on the indices `1:50`. Look up the functions `%in%` and `which` for ways to use this to get your subsets.

```
setosa = iris[iris$Species=="setosa",]
versicolor = iris[iris$Species=="versicolor",]
virginica = iris[iris$Species=="virginica",]

I=split(iris, sample(1:5, nrow(iris), replace=T))
iris1 = I[1]; iris2 = I[2]; iris3 = I[3]; iris4 = I[4]; iris5 = I[5]
iris.5fold <- list(iris1, iris2, iris3, iris4, iris5)
```

Distributions

Needless to say, *R* is a language geared toward statistical analysis, and thus there are many probability distributions that are avaliable as built-in functions. 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.

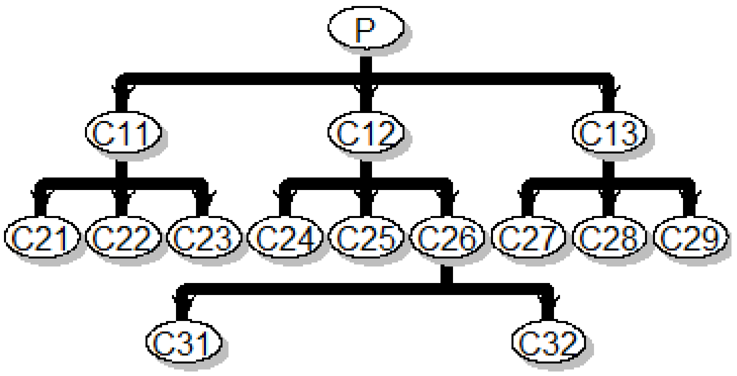
```r
plot(dnorm, xlim = c(-5, 5))   # bell curve of Normal density
plot(plogis, xlim = c(-5, 5))  # Logistic/Sigmoid function (CDF of Logistic d
istribution)
```

Appendix A

There are no exercises in this appendix. Its purpose is to provide more information about the topics covered in this assignment. Reading this material is encouraged, but not required.

Additional material on Lists

With lists, 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.



```r
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)
P   <- list( C11 = C11,
             C12 = C12,
             C13 = C13)

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

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

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

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

Additional material on Probability Distributions

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

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

```r
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)
```