SS2864B Assignment 1

Ali Al-Musawi

15/01/2020

Question 1:

To find all built-in functions related to the exponential distribution, use the following loop:

```
result <- c()
# loop for however many packages loaded.
for (i in 1:length(search())) {
  # add the relevent functions that contain the substring "exp" to the vector result.
  result <- c(result, ls(name = i, pattern = ".*exp.*"))
}
# display result
result
```

```
##  [1] "dexp"                    "expand.model.frame"
##  [3] "pexp"                    "qexp"
##  [5] "rexp"                    "SSbiexp"
##  [7] "as.expression"           "as.expression.default"
##  [9] "char.expand"             "exp"
## [11] "expand.grid"             "expm1"
## [13] "expression"              "gregexpr"
## [15] "is.expression"           "path.expand"
## [17] "regexpr"
```

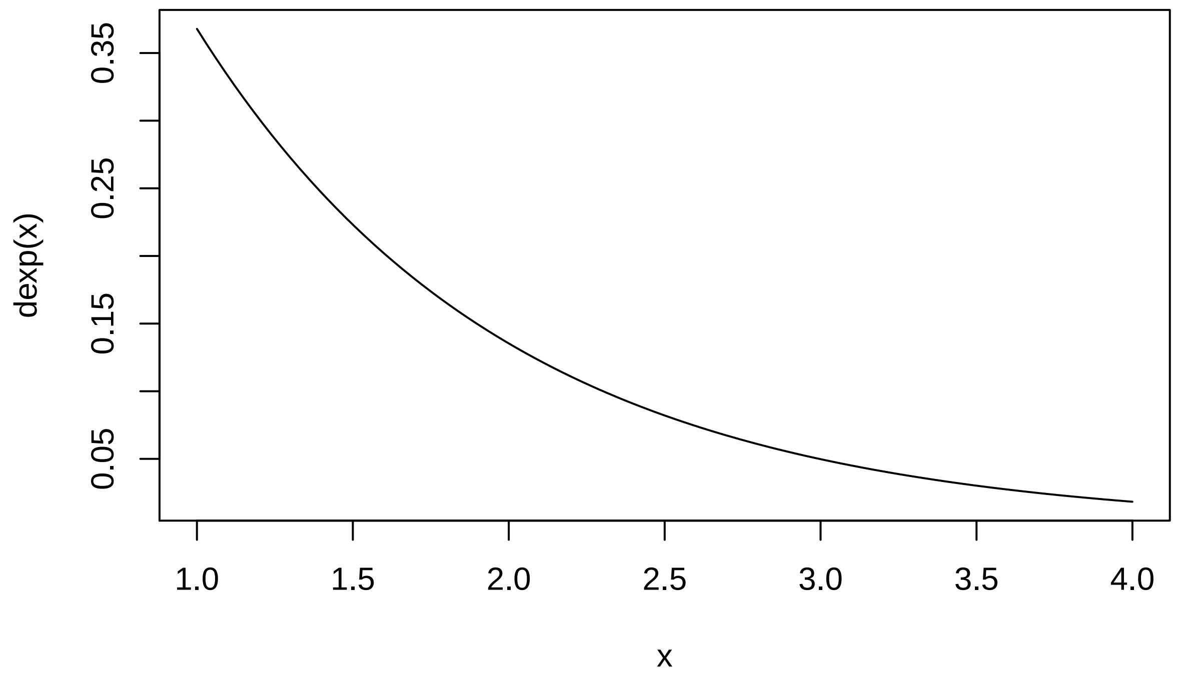
To refine the result, we notice that there are two keywords that got matched that we do not want: "expand" and "expression". As such, we use grep to get rid of such functions:

```
result <- c()
# loop for however many packages loaded.
for (i in 1:length(search())) {
    # add the relevent functions that contain the substring "exp" to the vector result.
  result <- c(result, ls(name = i, pattern = ".*exp.*"))
}
# discard any function name with substrings "expr" or "expand".
result = result[grep(pattern = ".*expr.*|.*expand.*", x = result, invert = TRUE)]
# display result
result
```

```
## [1] "dexp"    "pexp"    "qexp"    "rexp"    "SSbiexp" "exp"     "expm1"
```

The following function call produces a plot of the probability density function of the exponential distribution with the rate parameter λ set to 1 and the x-axis is restricted to the interval $[1, 4]$.

```
curve(dexp, from = 1, to = 4)
```



Question 2:

In this section, we compare codes defined by the user using traditional control flow mechanisms with R's built-in functions to compute the sum and mean of a vector of numeric data. Let x denote the list of numbers 1 through 100.

```r
x <- 1:100
# record the time since the code starts execution.
sumUserTime <- proc.time()
# s shall denote the partial sum in every iteration of the following loop.
s <- 0
for (i in x) {
  s <- s + i
}
# display the sum
s
```

```
## [1] 5050
```

```r
# subtract the initial time from the current time to find the execution duration.
sumUserTime <- proc.time() - sumUserTime
# measure the execution duration of R's built-in sum function to compare.
sumRTime <- proc.time()
# display the sum
sum(x)
```

```
## [1] 5050
```

```r
sumRTime <- proc.time() - sumRTime
sumUserTime
```

```
##    user  system elapsed
##    0.01    0.00    0.02
```

```r
sumRTime
```

```
##    user  system elapsed
##    0.02    0.00    0.01
```

```r
# compare the times recorded
print(paste(c("User sum implementation takes"), sumUserTime[3], "seconds.", collapse = ""))
```

```
## [1] "User sum implementation takes 0.0199999999999998 seconds."
```

```r
print(paste(c("R sum implementation takes"), sumRTime[3], "seconds.", collapse = ""))
```

```
## [1] "R sum implementation takes 0.01 seconds."
```

Next, we implement the arithmetic mean by modifying the code above and compare its performance with the R-defined mean function.

```r
x <- 1:100
# record the time since the code starts execution.
meanUserTime <- proc.time()
# s shall denote the partial sum in every iteration of the following loop.
s <- 0
for (i in x) {
  s <- s + i
}
m <- s/length(x)
# display the mean
m
```

```
## [1] 50.5
```

```r
# subtract the initial time from the current time to find the execution duration.
meanUserTime <- proc.time() - meanUserTime
# measure the execution duration of R's built-in mean function to compare.
meanRTime <- proc.time()
# display the sum
mean(x)
```

```
## [1] 50.5
```

```r
meanRTime <- proc.time() - meanRTime

# compare the times recorded
print(paste(c("User mean implementation takes"), meanUserTime[3], "seconds.", collapse = ""))
```

```
## [1] "User mean implementation takes 0.02 seconds."
```

```r
print(paste(c("R mean implementation takes"), meanRTime[3], "seconds.", collapse = ""))
```

```
## [1] "R mean implementation takes 0 seconds."
```

As we see, R functions take considerably less time compared to user-defined implementations to compute the sum and mean iteratively. This is because R's vectorized functions such as mean() and sum() are written in a compiled language such as C/C++, which takes much less time compared to our implementations in an interpreted language like R and Python. For each iteration, we have to check the data types we are processing, whereas in a compiled language with a vectorized setting, we only check the data types once.

Question 3:

In this section, we examine floating-point arithmetic overflow. Consider the following finite geometric summation:

$$\sum_{j=0}^{n} r^j$$

We are to evaluate it in three different appraoches. The first is using a for-loop, the second is using R's built-in sum() function, and the last using the closed formula for the geometric summation given by:

$$\sum_{j=0}^{n} r^j = \frac{1-r^{n+1}}{1-r}$$

Below, we set r to 1.08 and vary n.

```r
r <- 1.08
N <- c(10, 20, 30, 40)
# Approach 1: Using a For-Loop
for (n in N) {
  # let s denote the partial sum in every iteration.
  s <- 0
  for (j in 0:n) {
    s <- s + r^j
  }
  # display the result
  print(paste(c("sum of r^j from j = 0 to ", n,  " yields ", s), collapse = ""))
}
```

```
## [1] "sum of r^j from j = 0 to 10 yields 16.6454874631826"
## [1] "sum of r^j from j = 0 to 20 yields 50.4229214419656"
## [1] "sum of r^j from j = 0 to 30 yields 123.345868002491"
## [1] "sum of r^j from j = 0 to 40 yields 280.781040206798"
```

```r
# Approach 2: Using R's Built-in Function
for (n in N) {
  # generate vector
  x <- r^(0:n)
  # display the result
  print(paste(c("sum of r^j from j = 0 to ", n,  " yields ", sum(x)), collapse = ""))
}
```

```
## [1] "sum of r^j from j = 0 to 10 yields 16.6454874631826"
## [1] "sum of r^j from j = 0 to 20 yields 50.4229214419656"
## [1] "sum of r^j from j = 0 to 30 yields 123.345868002491"
## [1] "sum of r^j from j = 0 to 40 yields 280.781040206798"
```

```r
# Approach 3: Using a Closed-Form Formula
geomSum <- function(r, n) {
  return((1-r^(n+1))/(1-r))
}
for (n in N) {
  # display the result
  print(paste(c("sum of r^j from j = 0 to ", n,  " yields ", geomSum(r, n)), collapse = ""))
}
```

```
## [1] "sum of r^j from j = 0 to 10 yields 16.6454874631826"
## [1] "sum of r^j from j = 0 to 20 yields 50.4229214419656"
## [1] "sum of r^j from j = 0 to 30 yields 123.345868002491"
## [1] "sum of r^j from j = 0 to 40 yields 280.781040206798"
```

There is no apparent variation between the result of the approaches. However, theoretically, the last approach yields the most precise results. Care must be taken when manipulating floating-point numbers as repeated summation increases the error significantly. In fact, we can demonstrate the accuracy loss for a very large summation by setting $n = 1000$.

```r
r <- 1.08
n <- 1000
x <- r^(0:n)
error <- (1-r^(n+1))/(1-r) - sum(x)
# display the error
error
```

```
## [1] -4.611686e+18
```

The error is astronomical!

Question 4:

Let X be a standard normal random vector with n random variables. We want to confirm the empirical rule, which states that approximately 95% of the components of X have an absolute value less than two. In other words:

$$Pr(-2 < X_i < 2) \approx 0.95, 1 \leq i \leq n$$

```r
# set the size of the random vector.
n <- 1000
# test if the proportion is close 0.95 for different trials
for (i in 1:10) {
  # create a standard normal random vector of size n.
x <- rnorm(n)
# find the proportion of random variables whose absolute value is bounded by 2.
print(mean(abs(x)<2))
}
```

```
## [1] 0.96
## [1] 0.952
## [1] 0.957
## [1] 0.957
## [1] 0.953
## [1] 0.942
## [1] 0.951
## [1] 0.966
## [1] 0.951
## [1] 0.946
```

```r
# Repeat the same process for exponentially larger sample sizes:
for (i in 1:6) {
  x <- rnorm(10^i)
  print(mean(abs(x)<2))
}
```

```
## [1] 1
## [1] 0.99
## [1] 0.958
## [1] 0.9539
## [1] 0.95499
## [1] 0.95459
```

Question 5:

To create such the design matrix of a full 2^3 factorial design, we break the task to column by column.

```
# create an empty matrix object
M <- matrix()
# construct the matrix column by column
M <- cbind(1:8, rep(1:2, each = 4), rep(1:2, each = 2), rep(1:2, each = 1))
# display the result
M
```

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

Question 6:

In general, to obtain documentation in R, we call the help function.

```
help("cars")
```

```
## starting httpd help server ... done
```

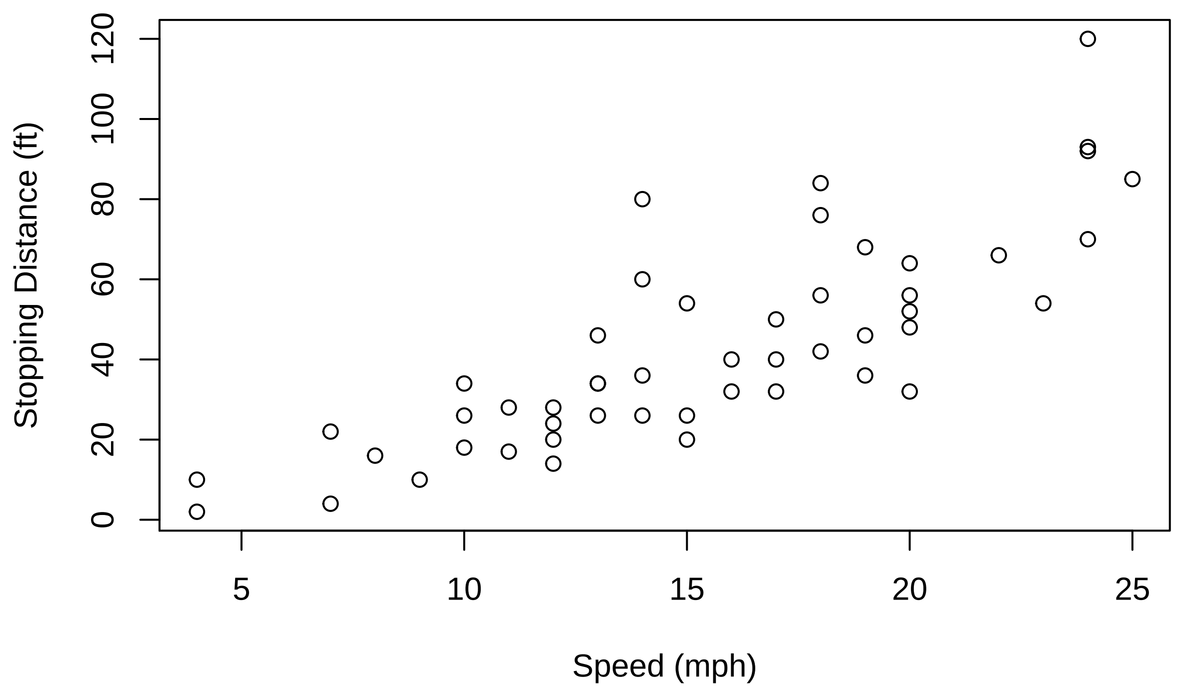
This opens up an html with details on usage, format, and description of the "cars", a dataset. According to the documentation, we have 50 observations and two variables: "speed" and "dist". To calculate the mean stopping distance for all observations for which the speed was 20 miles per hour, we need to subset the "dist" column and take the mean of the subsetted vector.

```
mean(cars$dist[cars$speed == 20])
```

```
## [1] 50.4
```

Finally, let us construct a scatter plot to relate the distance to speed.

```
plot(x=cars$speed, y=cars$dist, xlab="Speed (mph)", ylab="Stopping Distance (ft)")
```



The scatter plot appears to model the two variables in an increasing quadratic model when the stopping distance is considered a function of the speed. However, this is merely a hypothesis that needs to be tested using an appropriate regression inference test.

Question 7

Now, we consider another dataset: "USArrests". To extract the dimensions of this dataset, we use the dim() function, which returns a vector with the first entry being the number of rows and the second entry being the number of columns.

```
dim(USArrests)
```

```
## [1] 50  4
```

Therefore, there are 50 rows (observations) and 4 columns (variables). To compute the median of each column, we call a function that re-uses the median function for every column to obtain a list of medians.

```
lapply(USArrests, median)
```

```
## $Murder
## [1] 7.25
##
## $Assault
## [1] 159
##
## $UrbanPop
## [1] 66
##
## $Rape
## [1] 20.1
```

Let us see the effect of percent urban population on the average murder arrests per capita. We will examine the murder arrests rate when the percent urban population is less than 50 and when it is greater than 77.

```
# compute the average murder arrests per capita
m50 <- mean(USArrests$Murder[USArrests$UrbanPop<50])
m77 <- mean(USArrests$Murder[USArrests$UrbanPop>77])
# display the difference between the means
m77 - m50
```

```
## [1] 0.25
```

Note how the increase in percent urban population is met by increase in average murder arrests. This is not a coincidence. Finally, we construct a random sample without replacement from the records of USArrests dataset of size 12. To do, this we select a sample of the indices of the dataset to allow us to extract the entire set of records we are interested in, not just a specific variable.

```
# select a random sample of USArrests indices
ind <- sample(x=1:dim(USArrests)[1], replace = FALSE, size = 12)
USArr.Sample <- USArrests[ind,]
# display the sample dataset.
USArr.Sample
```

```
##                Murder Assault UrbanPop Rape
## Colorado          7.9     204       78 38.7
## New Hampshire     2.1      57       56  9.5
## Mississippi      16.1     259       44 17.1
## Washington        4.0     145       73 26.2
## Idaho             2.6     120       54 14.2
## Wyoming           6.8     161       60 15.6
## Oklahoma          6.6     151       68 20.0
## Alabama          13.2     236       58 21.2
## Maryland         11.3     300       67 27.8
## Indiana           7.2     113       65 21.0
## Hawaii            5.3      46       83 20.2
## Montana           6.0     109       53 16.4
```