STAT 782 Assignment 1

Francis Tang UPI: ftan638

Due: 27 March 2019

1. Use seq(), rep() and/or other commonly-used operators/functions and/or the recycling rule, but definitely not c(), nor any explicit loop, to create the following sequences.

```
0:23*15
```

```
##  [1]   0  15  30  45  60  75  90 105 120 135 150 165 180 195 210 225 240
## [18] 255 270 285 300 315 330 345
```

```
2^(0:8)
```

```
## [1]   1   2   4   8  16  32  64 128 256
```

```
rep(1:4, 4:1)
```

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

```
rep(0:3, 4) + rep(1:4*10, each=4)
```

```
##  [1] 10 11 12 13 20 21 22 23 30 31 32 33 40 41 42 43
```

```
paste(substring(rep("-", 6), first = 1, last = 0:1), "x^", 1:6, "/", 1:6, sep = "", collapse = " + ")
```

```
## [1] "x^1/1 + -x^2/2 + x^3/3 + -x^4/4 + x^5/5 + -x^6/6"
```

2. This question makes use of data on 10,000 electric scooter trips in Austin, Texas.1 Download the file "AustinDockless.csv" from Canvas and run the following code to load the data into an R data frame.

```
 dockless <- read.csv("AustinDockless.csv")
```

Find the length of the ten longest trips.

```
sort(dockless$Trip.Distance, decreasing=TRUE)[1:10]
```

```
##  [1] 19355 19263 19038 18394 14507 14451 12779 12601 12472 12263
```

Find the start and end Cell IDs for trips longer than 19km.

```
subset(dockless, Trip.Distance > 19000, select = Origin.Cell.ID:Destination.Cell.ID)
```

```
##      Origin.Cell.ID Destination.Cell.ID
## 4669          14074               14074
## 9518          15019               14227
## 9621          15019               14228
```

Find the longest trip (distance) between midnight and 8am.

```
# pick the 12nd and 13rd integer out from Start.Time as Hour
startHour = as.numeric(substring(dockless$Start.Time, 12, 13))

# pick the 21st and 22nd element out from Start.Time as AM/PM
startAMPM = substring(dockless$Start.Time, 21, 22)

# pick time between midnight and 8am
isRightStartPeriod = (startHour < 8 | startHour == 12) & startAMPM == "AM"

# same procedure for End.Time
```

```r
endHour = as.numeric(substring(dockless$End.Time, 12, 13))
endAMPM =  substring(dockless$End.Time, 21, 22)
isRightEndPeriod = (endHour < 8 | endHour == 12) & endAMPM == "AM"

# output the longest trip using max()
max(dockless[isRightStartPeriod & isRightEndPeriod,]$Trip.Distance)
```

```
## [1] 11977
```

Find the average speed and the distance covered for the top ten fastest trips (trips with highest average speed), where average speed is calculated as distance divided by duration.

```r
dockless$Speed <- (dockless$Trip.Distance / dockless$Trip.Duration)
dockless[order(dockless$Speed, decreasing=TRUE)[1:10],c(5,21)]
```

```
##      Trip.Distance     Speed
## 8909          6145 53.903509
## 1799           356 19.777778
## 9885          1286 14.613636
## 3674          5333 11.346809
## 978            357 10.818182
## 193            110 10.000000
## 4327           645  9.626866
## 5917          3440  8.775510
## 5982          3055  8.415978
## 4776          1245  8.190789
```

3. This question also works with the electronic scooter data. We will just focus on the distance variable Trip.Distance.

Calculate the mean and standard deviation of the distance data.

```r
distMean = mean(dockless$Trip.Distance)
print(distMean)
```

```
## [1] 1615.266
```

```r
distSD = sd(dockless$Trip.Distance)
print(distSD)
```

```
## [1] 1674.185
```

Calculate the median and upper and lower quartiles for the distance data (HINT: use the quantile() function).

```r
distQuartiles = quantile(dockless$Trip.Distance)
print(distQuartiles)
```

```
##       0%      25%      50%      75%     100% 
##     0.00   491.75  1142.00  2182.50 19355.00
```

Generate 10,000 random values from a Normal distribution (HINT: use the rnorm() function) with the same mean and standard deviation as the distance data and calculate the median and upper and lower quartiles of these random values.

```r
set.seed(1234)
rand = rnorm(10000, mean = distMean, sd = distSD)
randQuartiles = quantile(rand)
print(randQuartiles)
```

```
##         0%         25%         50%         75%        100%
## -4070.3709   508.9781  1623.0258  2736.4287  7672.6443
```

Generate 1,000,000 random values from a Normal distribution with the same mean and standard deviation as the distance data and, dealing with each consecutive subset of 10,000 values, calculate 100 upper quartiles (one for each of the 100 subsets of 10,000 values)

```
set.seed(1234)
randNew = rnorm(1000000, mean = distMean, sd = distSD)

# build a matrix based on randNew
matrixRandNew = matrix(randNew, ncol = 100)
Q3 = apply(matrixRandNew, 2, quantile)["75%",]
print(Q3)
```

```
##   [1] 2736.429 2748.037 2734.750 2778.251 2727.928 2747.678 2737.341
##   [8] 2733.882 2736.504 2772.283 2747.166 2740.335 2737.877 2749.035
##  [15] 2742.073 2747.603 2731.202 2745.210 2762.028 2722.479 2755.105
##  [22] 2784.287 2770.057 2744.672 2788.803 2732.448 2719.164 2716.093
##  [29] 2716.177 2748.662 2796.661 2770.102 2739.757 2696.621 2745.113
##  [36] 2758.199 2774.130 2745.969 2738.225 2766.942 2755.778 2757.481
##  [43] 2755.060 2768.428 2761.173 2745.528 2742.979 2721.388 2759.575
##  [50] 2750.646 2753.603 2747.491 2757.320 2770.252 2740.586 2739.986
##  [57] 2727.110 2787.447 2729.941 2744.196 2699.110 2777.860 2740.209
##  [64] 2734.215 2767.500 2755.582 2746.498 2755.516 2749.469 2772.825
##  [71] 2741.689 2695.096 2733.864 2763.335 2707.142 2758.902 2766.222
##  [78] 2740.095 2752.072 2726.040 2753.004 2747.965 2684.081 2757.887
##  [85] 2710.427 2748.324 2729.187 2781.094 2770.067 2735.959 2733.398
##  [92] 2750.425 2710.730 2767.942 2697.985 2764.782 2704.821 2728.067
##  [99] 2786.141 2723.497
```

What proportion of the upper quartiles from the random data are less than the distance upper quartile ?

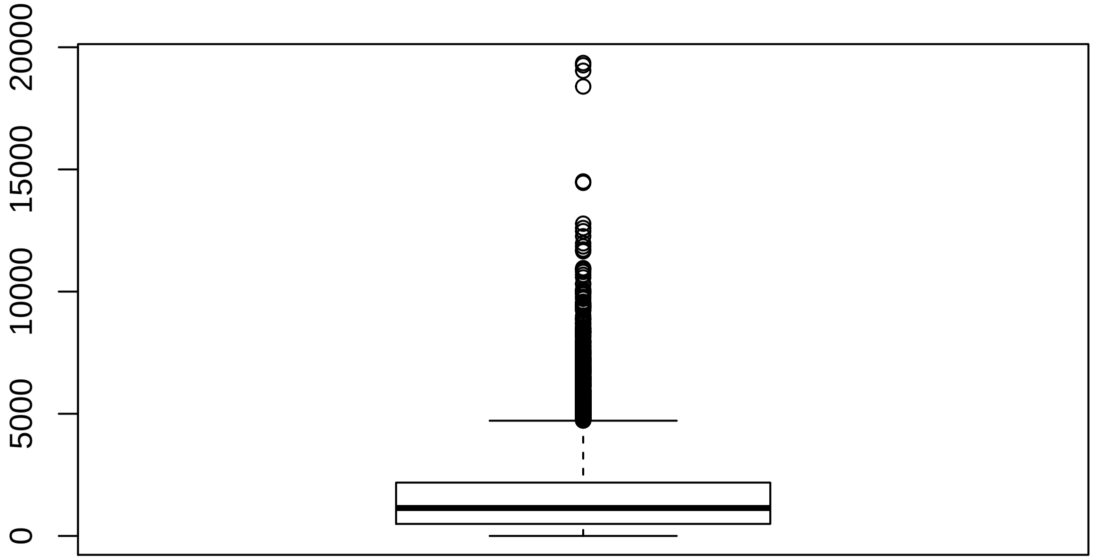
```
pval = (sum(Q3 < distQuartiles["75%"]))/length(Q3)
print(pval)
```

```
## [1] 0
```

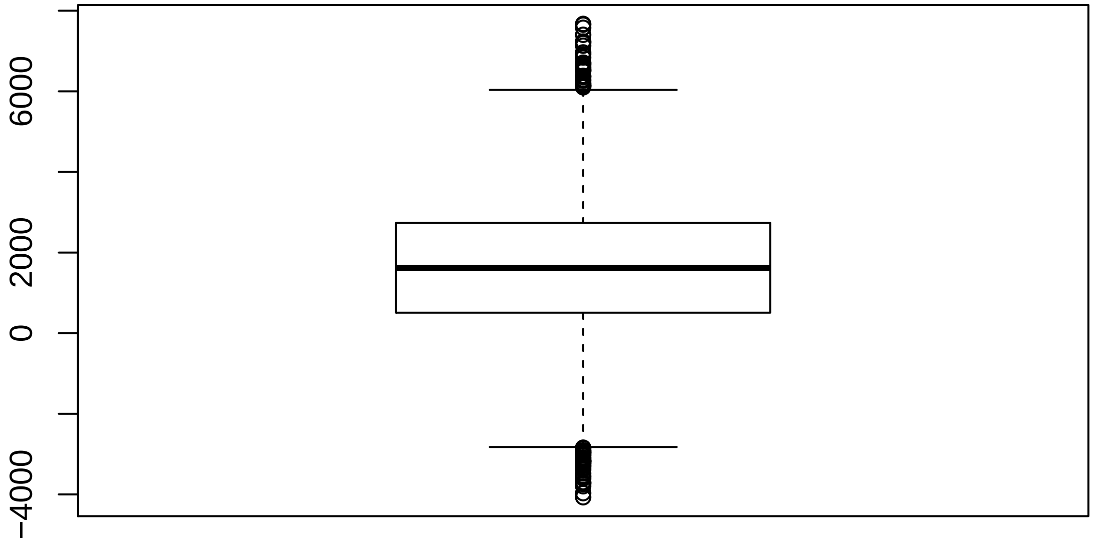
What does that tell us ?

Answer: From the boxplots below and histgrams shown in the question, we can figure out the following features of the dataset: 1. The original Trip.Distance is seriously right-skewed. 2. Most of the original data gathers around the range of [0,2500] and it has many outliars which spreads between [5000,20000]. This cause the original data has a much smaller median 1142 than the randomly generated data median 1623 which is the same as mean. 3. Because of the dense distribution between [0,2500] we mentioned in the original dataset, the upper quartile value from original distance 2182 is much smaller than the random data upper quartile value 2736.

```
boxplot(dockless$Trip.Distance)
```



```
boxplot(rand)
```



4. Write a function called segments that takes three arguments: from, to, and rep. The function should generate a matrix with two columns, the first column containing from values and the second column containing to values. The argument rep should default to FALSE.

```
segments = function(from, to, rep = FALSE){
  cbind(from,to)
}
```

Simplest use:

```
segments(.1, .2)
```

```
##      from  to
## [1,]  0.1 0.2
```

Write your function so that it satisfies all of the features described below. You should add one feature at a time and check that all previous features remain satisfied when you add a new feature. You should make use of vectorised operators and functions rather than for loops whenever possible.

The values in from and to should all be between -1 and 1. Values outside that range should be "clamped" to 1 with a warning.

```
segments = function(from, to, rep = FALSE){
  if(from < -1){
    warning("'from' value(s) less than -1 raised to -1 \n")
    from = -1
  }
  if(from > 1){
    warning("'from' value(s) larger than 1 reduced to 1 \n")
    from = 1
  }
  if(to < -1){
    warning("'to' value(s) less than -1 raised to -1 \n")
    to = -1
  }
  if(to > 1){
    warning("'to' value(s) larger than 1 reduced to 1 \n")
    to = 1
  }
  cbind(from,to)
}
```

Sample tests:

```
segments(.1, 1.2)
```

```
## Warning in segments(0.1, 1.2): 'to' value(s) larger than 1 reduced to 1

##      from to
## [1,]  0.1  1
```

```
segments(-1.2, .1)
```

```
## Warning in segments(-1.2, 0.1): 'from' value(s) less than -1 raised to -1

##      from  to
## [1,]   -1 0.1
```

The values in from and to can be vectors and they should both "recycle" to the longest length.

```
# this step will be updated with the next condition:
# by adding a validation function to bound values into [-1,1]
```

The rows of the matrix should be in order from smallest from value to largest from value.

```
segments = function(from, to, rep = FALSE){
  # validate all out-of-rage value(s) into [-1,1]

  if(any(from < -1)){
    warning("'from' value(s) less than -1 raised to -1 \n")
    from = -1
  }
  if(any(from > 1)){
    warning("'from' value(s) larger than 1 reduced to 1 \n")
    from = 1
  }
  if(any(to < -1)){
    warning("'to' value(s) less than -1 raised to -1 \n")
    to = -1
  }
  if(any(to > 1)){
    warning("'to' value(s) larger than 1 reduced to 1 \n")
    to = 1
  }

  # create matrix
  matrix = cbind(from,to)

  # sort from column from small to large
  matrix[order(matrix[,1]), ]

}
```

Tests:

```
segments(4:1/10, .9)
```

```
##      from  to
## [1,]  0.1 0.9
## [2,]  0.2 0.9
## [3,]  0.3 0.9
## [4,]  0.4 0.9
```

Tests:

```
segments(c(.1, .2), c(.3, .4))
```

```
##      from  to
## [1,]  0.1 0.3
## [2,]  0.2 0.4
```

```
segments(1:4/10, .9)
```

```
##      from  to
## [1,]  0.1 0.9
## [2,]  0.2 0.9
## [3,]  0.3 0.9
## [4,]  0.4 0.9
```

Any negative values in from and to should be treated as values measuring backward from 1.

```
segments = function(from, to, rep = FALSE){
  # validate all out-of-rage value(s) into [-1,1]

  if(any(from < -1)){
    warning("'from' value(s) less than -1 raised to -1 \n")
    from = -1
  }
  if(any(from > 1)){
    warning("'from' value(s) larger than 1 reduced to 1 \n")
    from = 1
  }
  if(any(to < -1)){
    warning("'to' value(s) less than -1 raised to -1 \n")
    to = -1
  }
  if(any(to > 1)){
    warning("'to' value(s) larger than 1 reduced to 1 \n")
    to = 1
  }
  # add 1 to all negative values
  from[from < 0] = 1 + from[from < 0]
  to[to < 0] = 1+ to[to < 0]

  # create matrix
  matrix = cbind(from,to)

  # sort from column from small to large
  matrix[order(matrix[,1]), ]

}
```

Tests:

```
segments(.1, -.1)
```

```
## from   to
##  0.1  0.9
```

```
segments(c(.1, .2), c(.7, -.2))
```

```
##      from  to
## [1,]  0.1 0.7
## [2,]  0.2 0.8
```

If any from values are larger than the corresponding to values, they should be swapped.

```
segments = function(from, to, rep = FALSE){
  # validate all out-of-rage value(s) into [-1,1]

  if(any(from < -1)){
    warning("'from' value(s) less than -1 raised to -1 \n")
    from = -1
  }
  if(any(from > 1)){
    warning("'from' value(s) larger than 1 reduced to 1 \n")
    from = 1
  }
  if(any(to < -1)){
    warning("'to' value(s) less than -1 raised to -1 \n")
    to = -1
  }
  if(any(to > 1)){
    warning("'to' value(s) larger than 1 reduced to 1 \n")
    to = 1
  }
  # add 1 to all negative values
  from[from < 0] = 1 + from[from < 0]
  to[to < 0] = 1+ to[to < 0]

  # create matrix
  matrix = cbind(from,to)

  # create boolean varible boollargerfrom
  boollargerfrom <- matrix[, "from"] > matrix[, "to"]
  # temporary variable set as "to"
  tempcol = matrix[, "to"][boollargerfrom]
  # if from is larger than to, start swapping
  matrix[, "to"][boollargerfrom] = matrix[, "from"][boollargerfrom]
  matrix[, "from"][boollargerfrom] = tempcol

  # sort from column from small to large
  matrix[order(matrix[,1]), ]

}
```

Tests:

```
segments(.9, .1)
```

```
## from   to
##  0.1  0.9
```

If rep is TRUE, the function should “repeat” the values in the argument until they exceed 1.

```
segments = function(from, to, rep = FALSE){
  # validate all out-of-rage value(s) into [-1,1]

  if(any(from < -1)){
    warning("'from' value(s) less than -1 raised to -1 \n")
    from = -1
  }
  if(any(from > 1)){
    warning("'from' value(s) larger than 1 reduced to 1 \n")
    from = 1
  }
  if(any(to < -1)){
    warning("'to' value(s) less than -1 raised to -1 \n")
    to = -1
  }
  if(any(to > 1)){
    warning("'to' value(s) larger than 1 reduced to 1 \n")
    to = 1
  }
  # add 1 to all negative values
  from[from < 0] = 1 + from[from < 0]
  to[to < 0] = 1+ to[to < 0]

  # create matrix
  matrix = cbind(from,to)

  # create boolean varible boollargerfrom
  boollargerfrom <- matrix[, "from"] > matrix[, "to"]
  # temporary variable set as "to"
  tempcol = matrix[, "to"][boollargerfrom]
  # if from is larger than to, start swapping
```

```r
  matrix[, "to"][boollargerfrom] = matrix[, "from"][boollargerfrom]
  matrix[, "from"][boollargerfrom] = tempcol

  # sort from column from small to large
  matrix = matrix[order(matrix[,1]), ]

  # when rep = TRUE, start the repeating process
  if(rep & matrix[length(matrix)] < 1.0){
    # if the initial matrix only has a single row
    if(length(matrix) == 2)
    {
      # calculate the gap value between every rows
      gap = (matrix[2] - matrix[1]) * 2
      # 'to' will start from initial value
      # then adding a gap by each row until it reaches 1
      to = seq(matrix[2], to = 1, by = gap)
      # 'from' will start from initial value
      # then add a gap by each row until it fits the amount of 'to'
      from = seq(matrix[1], by = gap, length = length(to))
    }
    # if the initial matrix has more than one row
    else{
      expasionTo = seq(matrix[nrow(matrix), "to"] + 0.1, by = 0.1, to = 1)
      expasionFrom = seq(matrix[nrow(matrix), "from"] + 0.1, by = 0.1, length = length(expasionTo))
      # append 'to' and 'from' with expasions
      from = c(matrix[,"from"], expasionFrom)
      to = c(matrix[,"to"], expasionTo)
      # delete some rows
      for (x in c(3, 5, 7)) {
        from <- from[-x]
      }
      for (y in c(3, 5, 7)) {
        to <- to[-y]
      }
      # make sure the last row will be [1.0, 1.0]
      if (from[length(from)] != to[length(to)])
        from[length(from)+1] = 1.0
        to[length(to)+1] = 1.0
    }

    # regenerate the matrix
    matrix = cbind(from,to)
    dimnames(matrix) = list(NULL, c("from", "to"))
  }
  print(matrix)
}
```

Tests:

```r
segments(.1, .2, rep=TRUE)
```

```
##      from  to
## [1,]  0.1 0.2
## [2,]  0.3 0.4
## [3,]  0.5 0.6
## [4,]  0.7 0.8
## [5,]  0.9 1.0
```

```
segments(c(.3, .1), .2, rep=TRUE)
```

```
##      from  to
## [1,]  0.1 0.2
## [2,]  0.2 0.3
## [3,]  0.4 0.5
## [4,]  0.5 0.6
## [5,]  0.7 0.8
## [6,]  0.8 0.9
## [7,]  1.0 1.0
```