Clebson Cardoso Alves de Sá

2016751813

1. Geometric Mean

The function bellow computes the geometric mean of a vector $x = \{x_1, x_2, \ldots, x_n\}$. If any of the values within the vector are negative, the function gives a warning message and computes the geometric mean. To compute the geometric mean, the following equations is used:

$$f(x) = \left(\prod_{i}^{n} x_i \right)^{\frac{1}{n}}$$

The code is quite simple, firstly we evaluate if any value of the vector is negative. If that is the case a warning message is returned informing the problem. Secondly the built-in `prod` function is used to compute the product of all values. Finally, the obtained result is raised to the power of $\frac{1}{n}$. The code can be viewed bellow.

```
GeometricMean <- function(values){
  if(sum(values < 0) > 0)
    warning("Negative values found.\n")
  return(prod(values)**(1/length(values)))
}
```

Let's make a few test cases for the `GeometricMean` function. We are going to use as example three test cases obtained from the MathIsFun website. As can be verified, all outputs are correctly computed.

```
GeometricMean(c(2, 18))
```

```
## [1] 6
```

```
GeometricMean(c(10, 51.2, 8))
```

```
## [1] 16
```

```
GeometricMean(c(1, 3, 9, 27, 81))
```

```
## [1] 9
```

---

2. Building a New Dataset

Lets build a dataset using the random generation normal distribution with mean equal to 3 and default standard deviation 1. To do that, it's necessary to simple use the `rnorm` function and append the obtained results into a matrix. The code to that is as follows:

```
set.seed(123)  # set the random seed generator
data <- data.frame(matrix(rnorm(10000, mean=3),
                          ncol=25,
                          dimnames=list(NULL, paste("X", 1:25, sep="."))))
```

Now let's take a look at how the newly created data frame looks like, to do that let's print the first 5 lines of the data.

```
head(data, 5)
```

```
##        X.1      X.2      X.3      X.4      X.5      X.6      X.7      X.8
## 1 2.439524 2.926444 3.356283 3.619850 2.710977 2.488396 2.516865 3.316985
## 2 2.769823 1.831349 2.341990 2.242490 3.656513 3.236938 2.468653 1.898265
## 3 4.558708 2.365252 3.855202 3.851525 2.546002 2.458411 2.412315 1.569042
## 4 3.070508 2.971158 4.152936 2.252070 2.406135 4.219228 2.588302 4.892011
## 5 3.129288 3.670696 3.276275 3.630240 1.289620 3.174136 3.709186 3.397877
##         X.9     X.10      X.11     X.12     X.13      X.14      X.15
## 1 2.1576737 2.786377 3.1965498 2.967122 2.367286 4.0679372 1.4383109
## 2 3.1018881 4.197876 3.6501132 2.223993 3.109172 2.7321025 1.8569736
## 3 2.1020742 3.231803 3.6710042 3.355759 1.437443 4.0110056 2.2774034
## 4 4.3939254 2.497159 1.7158422 1.887191 2.959755 0.6935654 3.5258316
## 5 0.5134761 3.630457 0.9738904 6.445992 2.963670 1.9486077 0.6558564
##       X.16     X.17     X.18     X.19     X.20     X.21     X.22     X.23
## 1 2.300772 1.644381 3.189951 3.928233 3.885298 3.511000 3.524349 2.780681
## 2 3.996452 2.939954 2.044564 3.359090 3.492430 4.807993 2.257916 2.734346
## 3 2.307255 1.529092 3.021324 3.186757 4.747074 1.297385 4.071473 5.110953
## 4 2.896517 4.116706 1.821166 2.580155 2.738001 3.287449 1.709632 2.590986
## 5 3.603866 2.831853 1.918325 1.834020 3.499755 2.730886 5.098183 2.445536
##       X.24     X.25
## 1 2.795852 1.816543
## 2 1.028524 2.624198
## 3 2.846310 1.532842
## 4 5.297633 2.704659
## 5 2.866059 2.436854
```

Now that we know how the data looks like, we can play around with the data solving the following problems:

2.1 Geometric Mean

Compute the geometric mean function in each column of the data frame data. Let's use the let's use the function `GeometricMean` created on topic 1. As we can observe, a few outputs are warning messages, since there are negative values in a few of the column's.

```r
for (col_idx in 1:dim(data)[2]){
  print(GeometricMean(data[,col_idx]))
}
```

```
## [1] 2.845667
## [1] 2.790897
## [1] 2.851296
## [1] 2.832801

## Warning in GeometricMean(data[, col_idx]): Negative values found.

## [1] NaN
## [1] 2.746103
## [1] 2.784419
## [1] 2.778567

## Warning in GeometricMean(data[, col_idx]): Negative values found.

## [1] NaN
## [1] 2.79284
## [1] 2.745977

## Warning in GeometricMean(data[, col_idx]): Negative values found.

## [1] NaN
## Warning in GeometricMean(data[, col_idx]): Negative values found.

## [1] NaN

## Warning in GeometricMean(data[, col_idx]): Negative values found.

## [1] NaN

## Warning in GeometricMean(data[, col_idx]): Negative values found.

## [1] NaN

## Warning in GeometricMean(data[, col_idx]): Negative values found.

## [1] 2.662041
## [1] 2.809164

## Warning in GeometricMean(data[, col_idx]): Negative values found.

## [1] NaN

## Warning in GeometricMean(data[, col_idx]): Negative values found.

## [1] NaN

## Warning in GeometricMean(data[, col_idx]): Negative values found.

## [1] NaN
## [1] 2.861628

## Warning in GeometricMean(data[, col_idx]): Negative values found.

## [1] NaN
## [1] 2.74123
## [1] 2.769901

## Warning in GeometricMean(data[, col_idx]): Negative values found.

## [1] NaN
```

2.2 Standard Deviation

Lets compute the standard deviation for each column in the dataset. R already has a built-in function called `sd` that computes the standard deviation. Let's use a vector to hold the standard deviation for each column, and only after computing the value for each column print the standard deviation for each column. The code is shown bellow:

```r
standard_deviation = c()  # vectr
for(i in 1:dim(data)[2]){
  standard_deviation = c(standard_deviation, sd(data[,i]))
}

print(standard_deviation)
```

```
##  [1] 0.9690155 0.9942858 1.0265244 0.9535985 1.0595188 0.9242590 0.9875931
##  [8] 1.0230263 1.0128798 0.9767226 1.0015290 0.9875529 0.9551536 1.0161600
## [15] 1.0215442 1.0515548 0.9542127 0.9521075 0.9990029 1.0270038 1.0145608
## [22] 1.0415851 1.0495298 0.9816793 0.9824599
```

As we can observe above, each value is the standard deviation for a given column in the range[1, 25]. Therefore the standard deviation for the first column is 0.9690, for the second is 0.9942 and the column 25 has a standard deviation of 0.9824.

2.3 Total

Now let's compute the total sum for each column. Just like the previous question, we are going to use R summation function which is called `sum` to compute the total for all columns of the data frame. The values for each column will be held in a vector and printed out after the total for all columns are computed.

```
total_sum = c()
for (i in 1:dim(data)[2]){
  total_sum = c(total_sum, sum(data[,i]))
}
print(total_sum)
```

```
##  [1] 1206.710 1201.978 1217.516 1206.573 1225.816 1171.785 1193.783
##  [8] 1193.497 1216.312 1195.349 1182.128 1199.899 1196.164 1206.780
## [15] 1217.340 1172.573 1200.971 1193.690 1211.847 1193.476 1223.183
## [22] 1207.280 1184.018 1184.266 1173.348
```

Just like for the standard deviation, the first value is the total for the first column, the second for the second column and so-on.

The sum for the lines are just alike, where the first value is the sum for the first line, the second value for the second line and so-on.

```
total_sum = c()
for (i in 1:dim(data)[1]){
  total_sum = c(total_sum, sum(data[i,]))
}
print(total_sum)
```

```
##   [1] 71.73364 70.60361 73.35342 73.96852 71.67460 76.14345 65.39629
##   [8] 75.96749 73.51287 79.47043 73.15345 82.73630 77.67339 80.49848
##  [15] 73.27900 68.91386 71.45829 79.00501 76.69118 69.60750 76.34278
##  [22] 77.36234 75.68799 79.20634 76.36493 78.27347 81.68888 75.60300
##  [29] 72.88100 79.73088 71.98136 80.76480 75.08489 75.45573 76.93119
##  [36] 74.15900 79.46573 73.55780 75.48729 77.41151 74.07114 81.51457
##  [43] 73.54876 75.14144 68.75509 69.17768 78.26494 72.15556 68.85395
##  [50] 65.45385 75.58295 78.23490 77.66865 83.17302 73.39014 72.70125
##  [57] 78.10815 76.98738 82.92555 69.82887 84.20664 73.93729 75.91876
##  [64] 77.40798 69.05837 83.51544 81.04910 81.27475 69.41206 77.73100
##  [71] 75.87863 69.86391 77.87316 75.26007 74.92016 76.00941 88.86601
##  [78] 70.59576 74.64284 70.01823 68.11358 77.09428 72.71213 77.97831
##  [85] 65.89708 83.15414 72.62713 73.96180 77.63202 66.75590 80.06310
##  [92] 78.93250 75.91606 78.22708 81.90855 82.39194 68.59069 77.10215
##  [99] 76.69871 70.41250 65.07066 76.04592 74.15908 72.57552 72.01311
## [106] 69.57921 75.38655 78.10113 74.19970 77.80671 85.86954 75.26667
## [113] 77.36195 77.85859 71.49077 74.76485 73.00807 69.68185 76.45379
## [120] 78.28583 76.25250 69.12233 72.86178 77.50893 71.43179 71.71567
## [127] 78.19391 72.85089 75.45531 77.12904 73.83503 69.89949 63.51611
## [134] 72.69047 66.19634 66.10830 82.59611 72.42426 75.99455 73.76734
## [141] 74.85904 66.39025 67.76550 74.63023 70.63364 69.79341 72.78685
## [148] 74.46242 84.34682 72.99568 77.89080 77.93795 70.11394 75.69171
## [155] 76.10969 78.89158 70.50168 67.06586 77.07818 71.87449 70.10104
## [162] 80.27072 69.46465 78.42809 71.70296 75.37521 66.02430 77.51163
## [169] 78.13316 73.94206 75.69763 70.83616 67.35567 73.37899 69.75451
## [176] 71.98867 77.80254 73.49850 79.37859 79.35584 80.49988 82.99821
## [183] 77.35215 75.48457 74.87607 64.43570 76.45381 74.13091 78.91937
## [190] 74.21466 69.91311 68.93009 78.51615 71.08478 73.13909 72.45103
## [197] 75.83811 71.32611 75.78983 79.29639 84.09221 76.72629 73.09833
## [204] 72.78738 82.62864 80.70113 73.80800 77.27572 83.11552 74.40004
## [211] 75.37379 73.28344 70.73991 76.33130 82.60805 67.54770 74.66896
## [218] 67.17194 73.88401 80.33475 73.17903 75.92310 86.58944 68.63653
## [225] 71.57585 70.95196 75.88092 77.53899 79.67843 72.21263 75.07443
## [232] 72.01180 74.04625 64.91462 70.98416 73.02980 70.25443 83.27024
## [239] 81.14073 69.42591 81.63668 75.79878 79.91031 84.85886 83.57277
## [246] 71.40209 78.89849 69.02615 82.21303 74.09667 64.73639 75.06233
## [253] 88.90088 77.17156 79.95854 70.71221 79.62445 89.12127 77.17141
## [260] 73.92009 79.21889 71.91015 80.82222 77.83051 69.21746 73.06355
## [267] 75.32490 71.63382 82.13441 71.92787 74.23427 83.55470 77.68355
## [274] 72.17891 70.30805 79.66801 81.28541 65.53034 68.20921 82.08565
## [281] 79.72521 71.29862 70.81100 70.98015 77.20179 77.75201 67.93932
## [288] 76.54810 68.74708 78.46550 74.62249 71.35479 84.67325 77.36224
## [295] 77.21424 67.74225 77.48500 68.14004 72.27337 69.13095 76.43507
## [302] 68.31276 61.04682 69.47684 77.64809 77.91084 82.97840 73.77451
## [309] 73.23421 77.15837 76.03962 78.78756 75.57885 70.52875 67.93445
## [316] 79.87271 86.10361 64.54097 72.60440 76.67202 70.23970 78.23853
## [323] 74.93202 77.95747 70.55095 79.50450 83.11256 66.99315 70.87809
## [330] 69.01001 85.18413 70.97827 72.07389 65.42116 76.88236 77.28836
## [337] 74.70382 75.22717 71.60717 81.41069 71.43952 75.56236 76.78704
## [344] 75.94237 76.21864 68.01450 74.94975 74.41891 70.76781 74.96726
## [351] 79.65006 74.38799 75.00761 74.89413 76.81265 73.55249 80.90404
## [358] 79.47660 63.15220 77.91841 82.14016 76.31236 78.93734 77.89846
## [365] 74.14325 69.98949 82.64534 75.03965 79.41123 74.47072 71.16582
## [372] 58.62354 79.01555 79.57733 73.50159 86.67565 72.69331 77.48268
## [379] 79.03679 66.89161 72.32194 75.66032 74.12861 72.66777 68.41865
## [386] 80.34515 68.06160 69.65246 76.72094 77.10988 71.00221 74.08218
## [393] 80.95152 77.36456 68.49851 74.78358 76.12767 78.35537 75.80050
## [400] 73.51288
```

2.4 Conditional Selection

Now let's select a subset of the original data, considering the restriction that the values for the column $X.1 > 3$ and the values for the column $X.20 < 3$. There a few possibilities of doing this, the simpler way is to use the built-in R function `subset`, which return slices of the data that meet the giving condition. The code is shown bellow:

```
sub <- subset(data, data$X.1 > 3 & data$X.20 < 3)
dim(sub)  # return the dimension of the data frame (lines, columns)
```

```
## [1] 102  25
```

The second way is to use R data slicing, which is shown in the code bellow:

```
sub2 <- data[data$X.1 > 3 & data$X.20 < 3,]
dim(sub2) # return the dimension of the data frame (lines, columns)
```

```
## [1] 102  25
```

As it's possible to see, both approaches indicates that after meeting the condition, only 102 lines of the data are left.

2.5 Replacing Column Names

Let's take a look of how the column names looks like:

```
colnames(data)
```

```
##  [1] "X.1"  "X.2"  "X.3"  "X.4"  "X.5"  "X.6"  "X.7"  "X.8"  "X.9"  "X.10"
## [11] "X.11" "X.12" "X.13" "X.14" "X.15" "X.16" "X.17" "X.18" "X.19" "X.20"
## [21] "X.21" "X.22" "X.23" "X.24" "X.25"
```

To modify the column names, it's necessary to replace the current variable names using the function `colnames` by passing the new variable names. The fragment of code to do this is quite simple, and can be seen bellow:

```
new_col_names = sprintf("%s%d", "Var", 1:25)
colnames(data) <- new_col_names
head(data, 5)
```

```
##       Var1     Var2     Var3     Var4     Var5     Var6     Var7     Var8
## 1 2.439524 2.926444 3.356283 3.619850 2.710977 2.488396 2.516865 3.316985
## 2 2.769823 1.831349 2.341990 2.242490 3.656513 3.236938 2.468653 1.898265
## 3 4.558708 2.365252 3.855202 3.851525 2.546002 2.458411 2.412315 1.569042
## 4 3.070508 2.971158 4.152936 2.252070 2.406135 4.219228 2.588302 4.892011
## 5 3.129288 3.670696 3.276275 3.630240 1.289620 3.174136 3.709186 3.397877
##        Var9    Var10     Var11    Var12    Var13     Var14     Var15
## 1 2.1576737 2.786377 3.1965498 2.967122 2.367286 4.0679372 1.4383109
## 2 3.1018881 4.197876 3.6501132 2.223993 3.109172 2.7321025 1.8569736
## 3 2.1020742 3.231803 3.6710042 3.355759 1.437443 4.0110056 2.2774034
## 4 4.3939254 2.497159 1.7158422 1.887191 2.959755 0.6935654 3.5258316
## 5 0.5134761 3.630457 0.9738904 6.445992 2.963670 1.9486077 0.6558564
##      Var16    Var17    Var18    Var19    Var20    Var21    Var22    Var23
## 1 2.300772 1.644381 3.189951 3.928233 3.885298 3.511000 3.524349 2.780681
## 2 3.996452 2.939954 2.044564 3.359090 3.492430 4.807993 2.257916 2.734346
## 3 2.307255 1.529092 3.021324 3.186757 4.747074 1.297385 4.071473 5.110953
## 4 2.896517 4.116706 1.821166 2.580155 2.738001 3.287449 1.709632 2.590986
## 5 3.603866 2.831853 1.918325 1.834020 3.499755 2.730886 5.098183 2.445536
##      Var24    Var25
## 1 2.795852 1.816543
## 2 1.028524 2.624198
## 3 2.846310 1.532842
## 4 5.297633 2.704659
## 5 2.866059 2.436854
```

As it's possible to see in the head for the dataset, all column names were modified for the $Var1, Var2, \ldots, Var25$.

---

3. Iris Data Set

Reading the data set Iris and verifying the first 5 lines.

```
iris = read.csv("./iris.csv")
head(iris, 5)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
```

3.1 Obtaining help

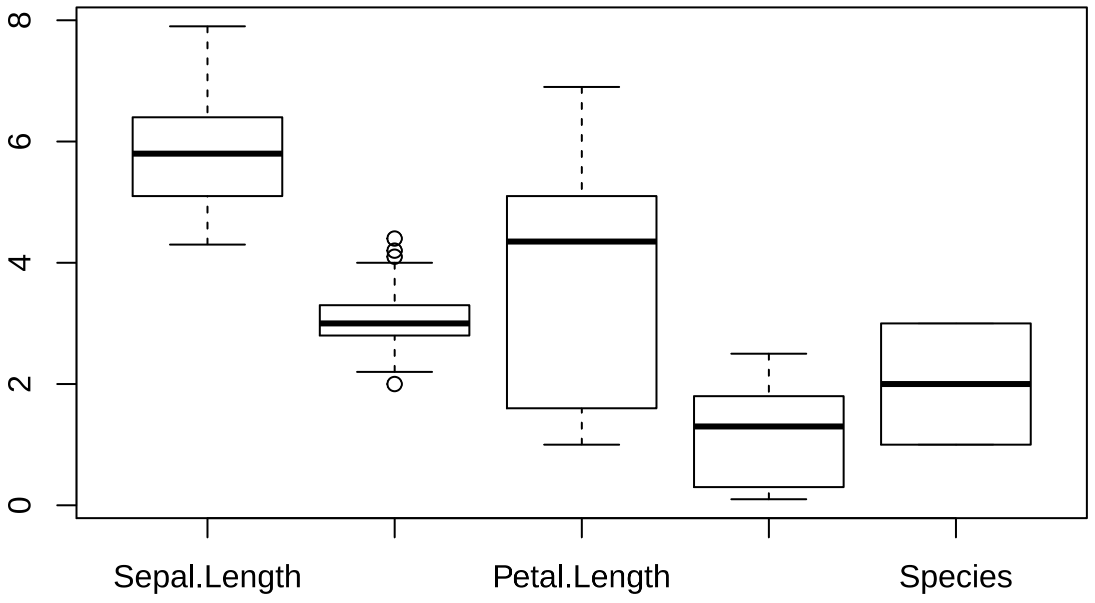
The documentation for datasets that are well known and the built-in functions in R can be checked using either `?` or the `help` function. This is done in the following way:

```
?iris
help("iris")
```

3.2 Boxplot

It's possible to plot a box-and-whisker plot by calling the `boxplot` function in R. If a dataframe is given, it plots all variables in a single plot. The code is shown bellow:

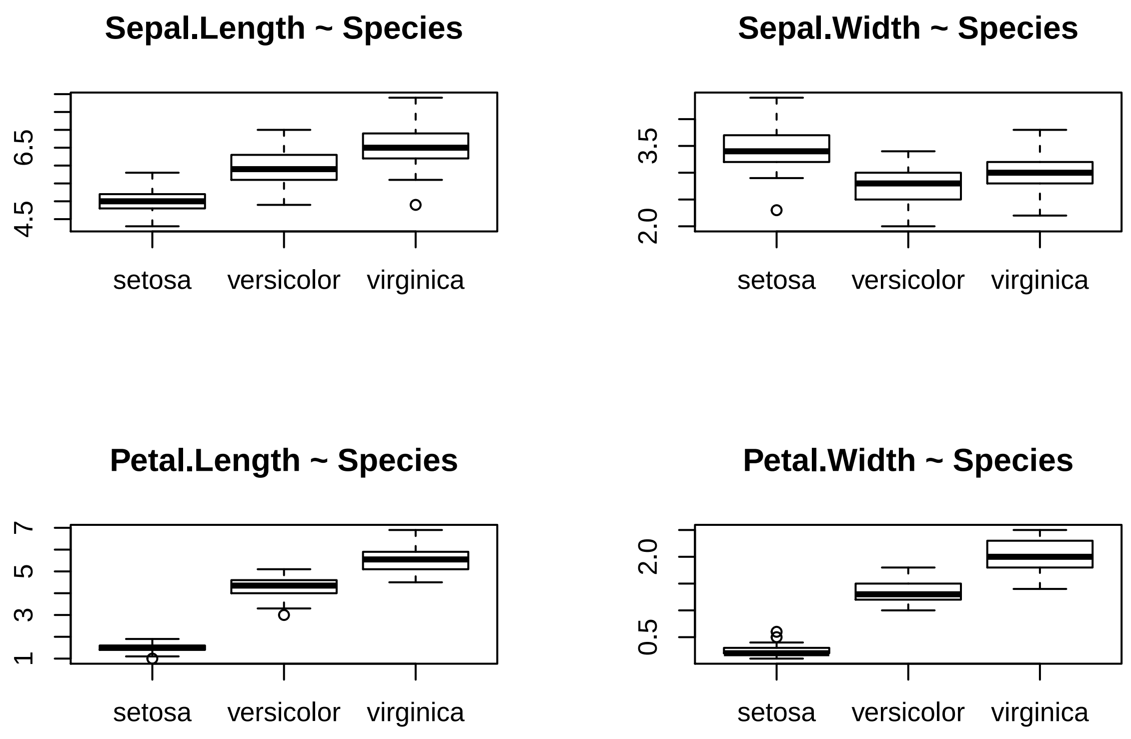
```
boxplot(iris)
```



3.3 BoxPlot in Groups

To plot multiple charts in the same frame we can split the window using the R function `par(mfrow=c(lines,cols))`. The aggregation of groups can be combined using the Tilde Operator `~` to separate the left and right hand sides of the dataframe. This operation is done for each attribute of the iris dataset grouped by the attribute Species.

```
par(mfrow=c(2,2))
boxplot(iris$Sepal.Length ~ iris$Species, main="Sepal.Length ~ Species")
boxplot(iris$Sepal.Width ~ iris$Species, main="Sepal.Width ~ Species")
boxplot(iris$Petal.Length ~ iris$Species, main="Petal.Length ~ Species")
boxplot(iris$Petal.Width ~ iris$Species, main="Petal.Width ~ Species")
```

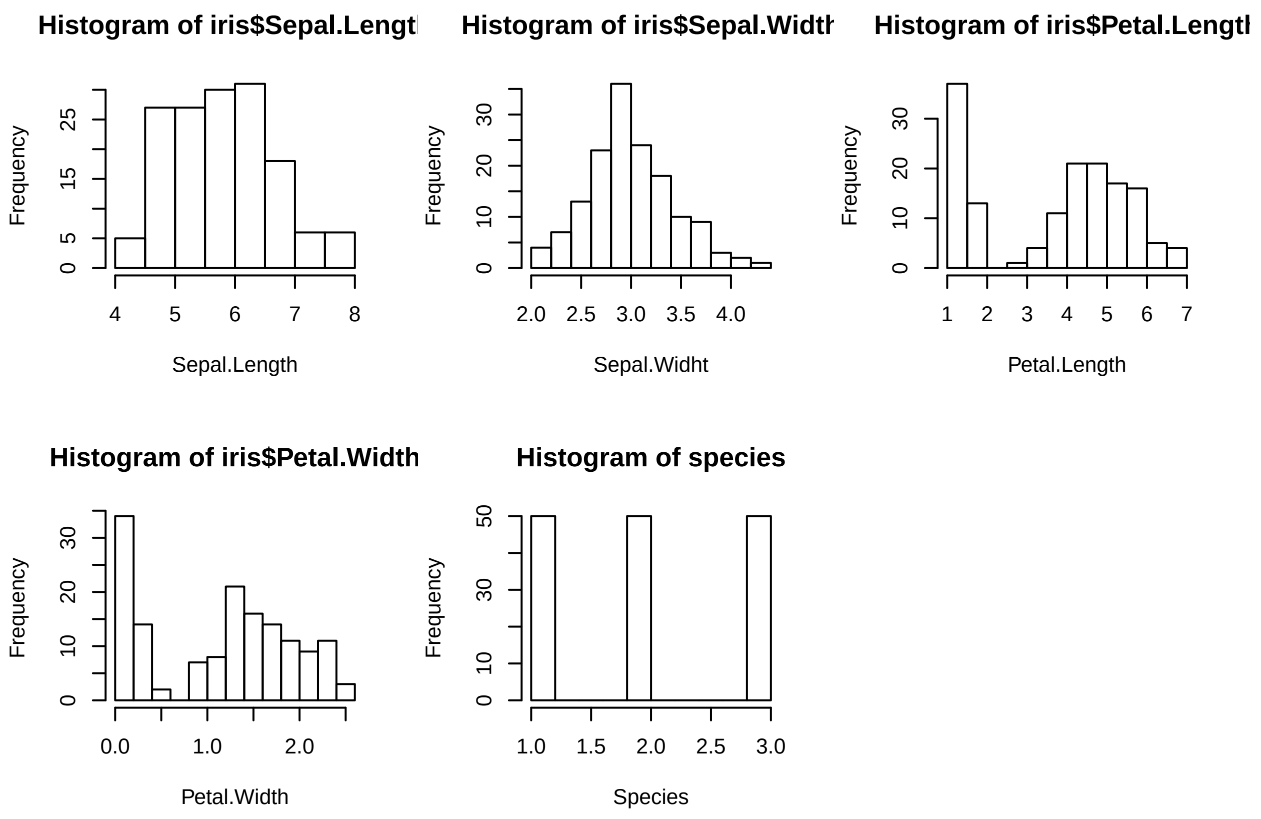


3.4 Histogram

To compute the histogram, R provides the function `hist`. In the following chart it's shown the histogram for each column of the data set iris. To plot the histogram for the column Species, was necessary to cast it's values to numeric.

```r
par(mfrow=c(2,3))
hist(iris$Sepal.Length, xlab = "Sepal.Length")
hist(iris$Sepal.Width, xlab = "Sepal.Widht")
hist(iris$Petal.Length, xlab = "Petal.Length")
hist(iris$Petal.Width, xlab = "Petal.Width")

species = sapply(iris$Species, as.numeric)  # return the column species as numeric
hist(species, xlab = "Species")
```



---

4 Attitude Dataset

Let's read the data set attitude and check-out the 5 first lines of the data.

```
attitude = read.csv('./attitude.csv')
head(attitude, n=5)
```

```
##   rating complaints privileges learning raises critical advance
## 1     43         51         30       39     61       92      45
## 2     63         64         51       54     63       73      47
## 3     71         70         68       69     76       86      48
## 4     61         63         45       47     54       84      35
## 5     81         78         56       66     71       83      47
```

4.1 Discovering the dataset

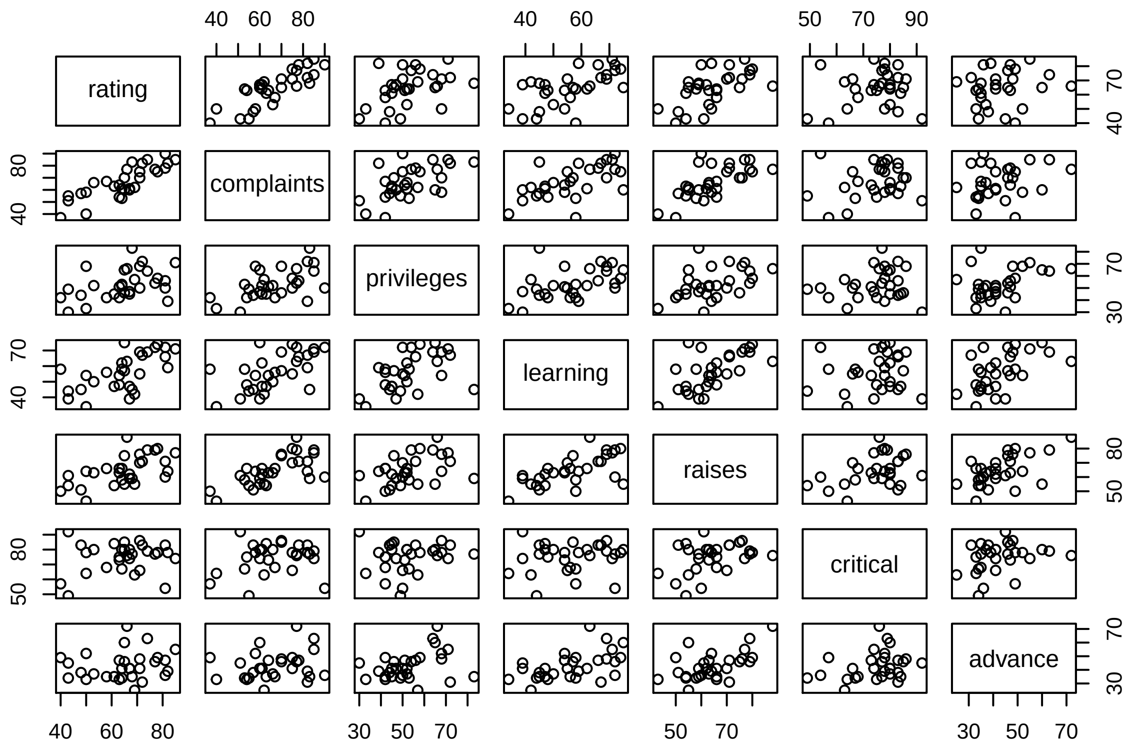
To visualize the dataset information we can check the R documentation in the same way we did for the Iris collection.

```
?attitude
help(attitude)
```

4.2 Plotting the Data

To plot a scatter plot of all attributes against all atributes we can call the `plot` function within R. In doing so, a $N \times N$ matrix of all atributes is ploted. The result of this plot can ve visualized bellow:

```
plot(attitude)
```



4.3 Computing the Mean

There are basically two ways of computing the mean of columns. One was already explained in exercises 2.2 (Standard Deviation) and 2.3 (Total Sum), which basically consists of looping through each column and then applying the function to be computed upon the rows. The code for this approach is shown bellow:

```
mean_list = c()
for(i in 1:dim(attitude)[2])
  mean_list = append(mean_list, mean(attitude[,i]))
print(mean_list)
```

```
## [1] 64.63333 66.60000 53.13333 56.36667 64.63333 74.76667 42.93333
```

A tider and simpler way of doing this consist of using the `sapply` function within R. This approach perform the same operation, but much more faster. The code can be seen bellow:

```
means_attitude <- sapply(attitude, mean)
print(means_attitude)
```

```
##     rating complaints privileges   learning     raises   critical
##   64.63333   66.60000   53.13333   56.36667   64.63333   74.76667
##    advance
##   42.93333
```

As can be seen, both operation are correctly computed with the same output for each column.

4.4 Transforming a variable

It's possible to easily transform the values of a column to discrete values with the command `cut`. This function basicaly divides the values in intervals considering the buckets given as slicing parameters. To update the dataframe, the only thing required is to make an assignment of the discrete values into the desired column of the dataframe. The code bellow shows how this procedure is done. The first 5 lines of the dataset is shown, and as we can observe, the column `attitude$complaints` is now categoric.

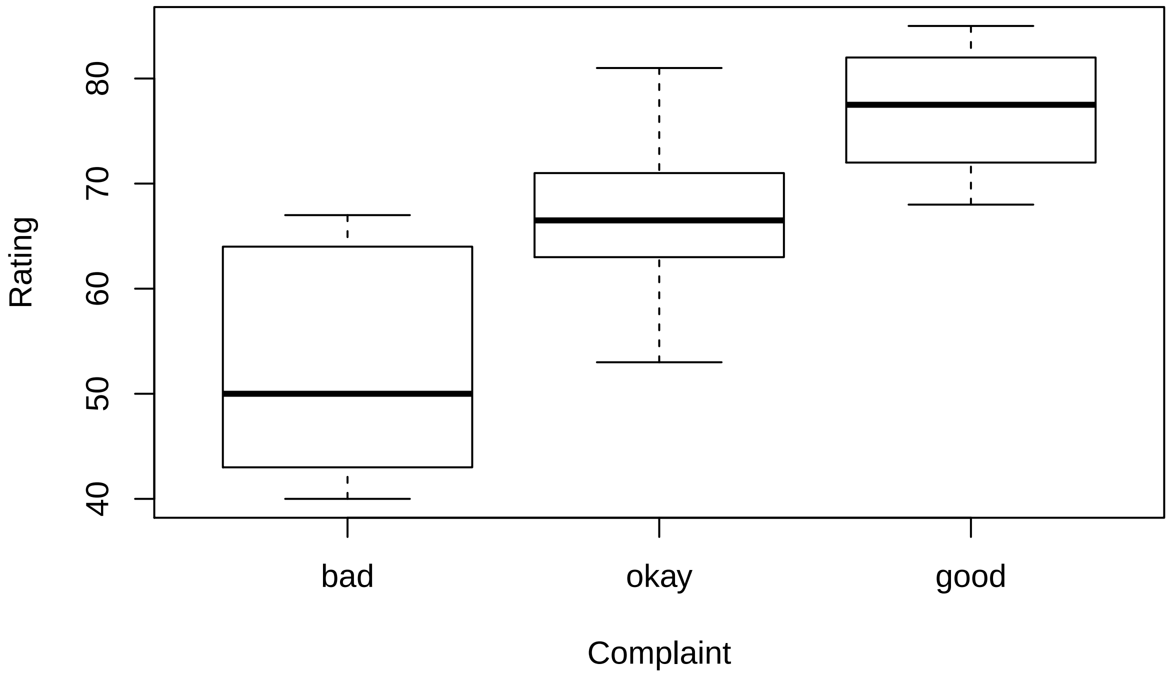
```
attitude$complaints = cut(attitude$complaints,
                          c(-1, 60, 80, 100),
                          labels=c("bad", "okay", "good"))
head(attitude, 5)
```

```
##   rating complaints privileges learning raises critical advance
## 1     43        bad         30       39     61       92      45
## 2     63       okay         51       54     63       73      47
## 3     71       okay         68       69     76       86      48
## 4     61       okay         45       47     54       84      35
## 5     81       okay         56       66     71       83      47
```

4.5 Rating ~ Complaints

Let's visualize the box-and-whisker plot by grouping the dataset by the categorical column `attitude$complaints`. As we can observe, 50% of the data representing *bad* complaints are between ($\approx 43, \approx 65$) with a median of 50. For the neutral complaints *okay*, the 50% quantile of the ratings are distributed between 60 and 71 with a median of ≈ 67. The ratings for complaints considered *good* are scattered with 50% of the data between ($\approx 71, \approx 80$) with a median of ≈ 77.

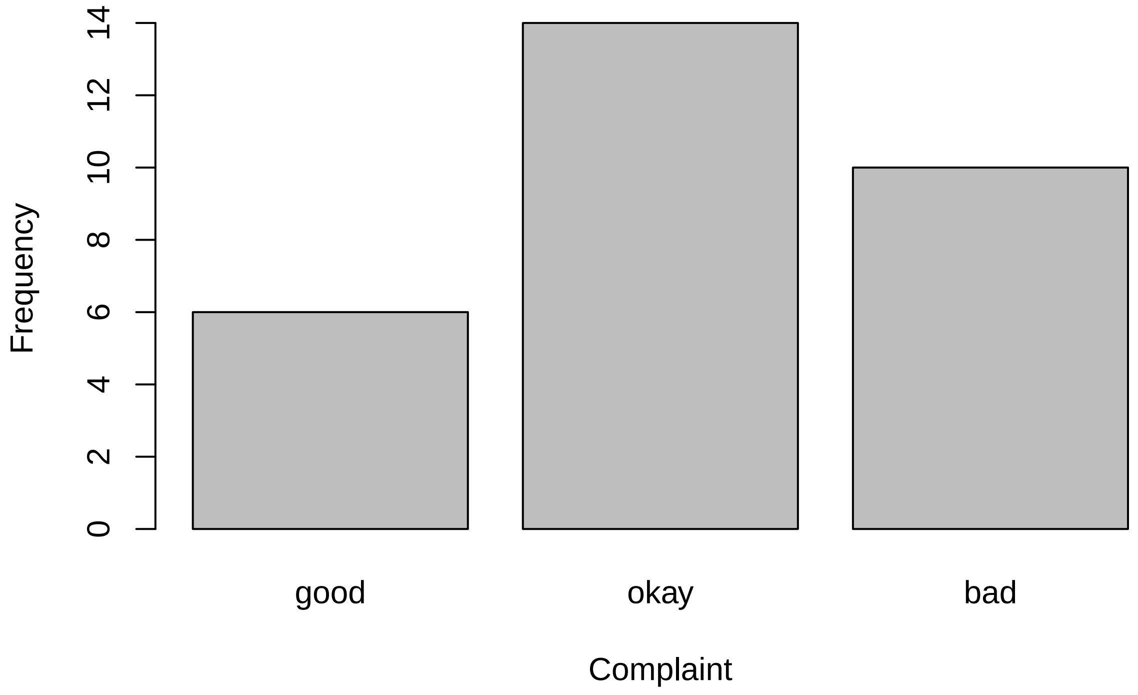
```
boxplot(attitude$rating ~ attitude$complaints, xlab="Complaint", ylab="Rating")
```



4.6 Bar Chart

Let's see the frequency of each complaint by using the bar chart. The first thing to do is to compute the amount of complaints for each category. This is done by using the logical operator of equivalence. Every outcome that is `TRUE` is considered to be one, while `FALSE` is considered to be zero. Therefore, to get the amount of complaints of each categorical value, we just need sum up the outcome of each comparison. After that, we just call the `barplot` function to plot the chart. This procedure is shown bellow.

```r
good <- sum(attitude$complaints=='good')
okay <- sum(attitude$complaints=='okay')
bad <- sum(attitude$complaints=='bad')
barplot(c(good, okay, bad),
        names.arg = c('good', 'okay', 'bad'),
        ylab='Frequency',
        xlab='Complaint')
```



4.7 Modyfying Variable Names

To modify the names of the discrete variable `attitude$complaints` it is necessary to replace the old values for the new categories. This can be done by calling the `factor` function. The code for this procedure is shown bellow. As it's possible to see for the first 5 lines, the collumn `attitude$complaints` contains the new categories.

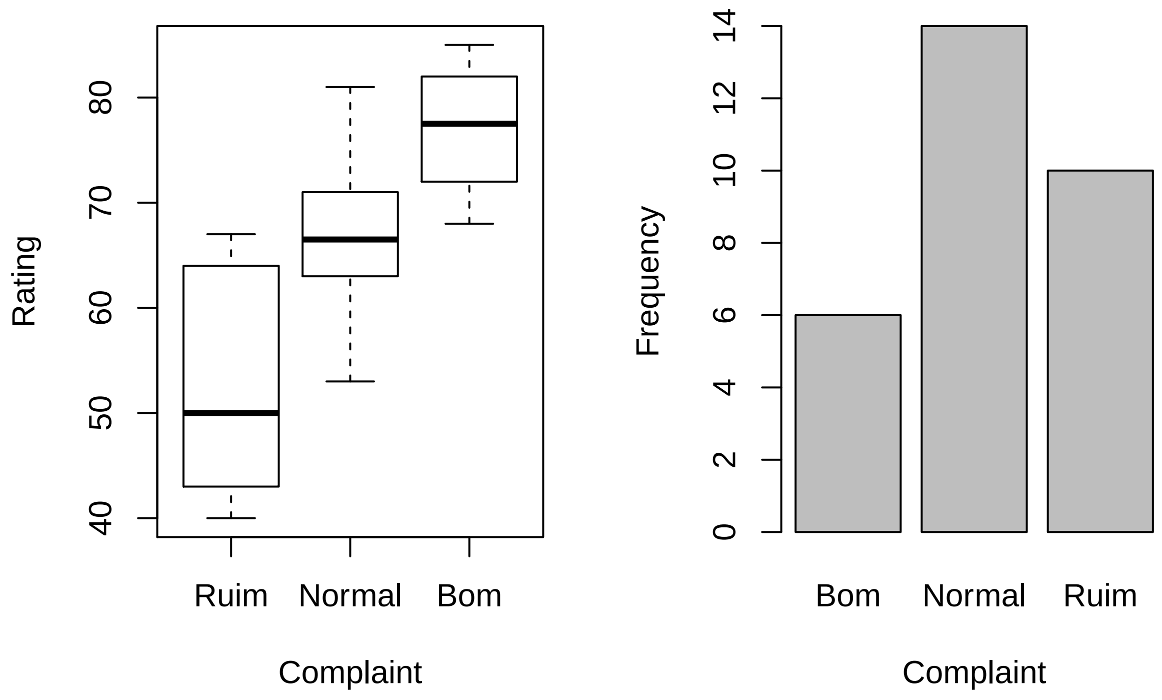
```r
attitude$complaints = factor(attitude$complaints, labels = c("Ruim", "Normal", "Bom"))
head(attitude, 5)
```

```
##   rating complaints privileges learning raises critical advance
## 1     43       Ruim         30       39     61       92      45
## 2     63     Normal         51       54     63       73      47
## 3     71     Normal         68       69     76       86      48
## 4     61     Normal         45       47     54       84      35
## 5     81     Normal         56       66     71       83      47
```

Now that the new categories were applied, the only thing remaining is to redo the plots considering the new labels for categorical variable.

```r
par(mfrow = c(1, 2))
boxplot(attitude$rating ~ attitude$complaints, xlab="Complaint", ylab="Rating")

good <- sum(attitude$complaints=='Bom')
okay <- sum(attitude$complaints=='Normal')
bad <- sum(attitude$complaints=='Ruim')
barplot(c(good, okay, bad),
        names.arg = c('Bom', 'Normal', 'Ruim'),
        ylab='Frequency',
        xlab='Complaint')
```


Rating
Ruim
Normal
Bom
Complaint
Frequency
Bom
Normal
Ruim
Complaint