Biostatistics using R

Dewey Brooke

2018-06-29

Contents

List of Tables

List of Figures

Preface

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.name/tinytex/.

1 Introduction

Reading Data Files into R

The first step in every analysis requires data to be read into the environment, and learning how to do this is the first hurdle a person needs to overcome to begin learning to use R.

Data can exist in many different formats, either as the generic universal types (e.g. csv, tsv, .json, etc) or software specific types (e.g. `.xlsx`, ")

In this chapter, we will first discuss how to read data using functions in Base-R (when possible), and then we will discuss alternative packages, such as the multitude of packages in the Tidyverse, and highlight their advantages over Base-R functions.

1.0.1 Generic Formats

1.0.1.1 CSV- Comma Separated Values

The fields are separated by a comma , and are typically used for loading into spreadsheets.

For example:

```
csv_example_path <- "data/ASCII-comma/FEV.DAT.txt"


readLines(csv_example_path)[1:8]  # reads each line of the file
```

```
[1] "'Id','Age','FEV','Hgt','Sex','Smoke'"
[2] "301,9,1.708,57,0,0"
[3] "451,8,1.724,67.5,0,0"
[4] "501,7,1.72,54.5,0,0"
[5] "642,9,1.558,53,1,0"
[6] "901,9,1.895,57,1,0"
[7] "1701,8,2.336,61,0,0"
[8] "1752,6,1.919,58,0,0"
```

```
# Note: readLines(csv_example_path) is the same as
# readLines("data/ASCII-comma/FEV.DAT.txt")
```

In Base-R, CSV data can be read using the `read.csv()` function. The `read.csv2()` function is used in countries that use a comma as a decimal point and a semicolon as a field separator.

```
csv_example <- read.csv(csv_example_path)

head(csv_example)
```

```
  X.Id. X.Age. X.FEV. X.Hgt. X.Sex. X.Smoke.
1   301      9  1.708   57.0      0        0
2   451      8  1.724   67.5      0        0
3   501      7  1.720   54.5      0        0
4   642      9  1.558   53.0      1        0
5   901      9  1.895   57.0      1        0
6  1701      8  2.336   61.0      0        0
```

1.0.1.2 TSV- Tab Separated Values

The fields are separated by a tabulation or ̂and are saved as `.txt` files. However, not all `.txt` files contain tab separated values.

For example:

```
tsv_example_path <- "data/ASCII-tab/FEV.DAT.txt"

readLines(tsv_example_path)[1:8]
```

```
[1] "'Id'\t'Age'\t'FEV'\t'Hgt'\t'Sex'\t'Smoke'"
[2] "301\t9\t1.708\t57\t0\t0"
[3] "451\t8\t1.724\t67.5\t0\t0"
[4] "501\t7\t1.72\t54.5\t0\t0"
[5] "642\t9\t1.558\t53\t1\t0"
[6] "901\t9\t1.895\t57\t1\t0"
[7] "1701\t8\t2.336\t61\t0\t0"
[8] "1752\t6\t1.919\t58\t0\t0"
```

```r
tsv_example <- read.delim("data/ASCII-tab/FEV.DAT.txt")
head(tsv_example)
```

```
  X.Id. X.Age. X.FEV. X.Hgt. X.Sex. X.Smoke.
1   301      9  1.708   57.0      0        0
2   451      8  1.724   67.5      0        0
3   501      7  1.720   54.5      0        0
4   642      9  1.558   53.0      1        0
5   901      9  1.895   57.0      1        0
6  1701      8  2.336   61.0      0        0
```

1.0.2 Excel

```r
library(readxl)
```

1.0.3 Software Specific Formats

R is increasingly recognized as the gold standard for statistical computations, yet some of your future collaborates will exclusively use Commercial Software (SAS, SPSS, Matlab, and Stata) for their statistical computations. Although these individuals are limited by the types of files they can read or write, the `haven` R-package can both read and write any of these file formats.

```r
library(haven)
```

1.0.3.1 SAS(`.sas7bdat`), SPSS(`.sav`,`.por`, `.xpt`), Stata (`.dta`)

```r
sas <- read_sas("data/SAS/FEV.sas7bdat")

head(sas)
```

```
# A tibble: 6 x 6
     ID   AGE   FEV   HGT   SEX SMOKE
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1   301     9  1.71  57       0     0
2   451     8  1.72  67.5     0     0
3   501     7  1.72  54.5     0     0
4   642     9  1.56  53       1     0
5   901     9  1.90  57       1     0
6  1701     8  2.34  61       0     0
```

```r
spss <- read_spss("data/SPSS/FEV.DAT.sav")
head(spss)
```

```
# A tibble: 6 x 6
     Id   Age   FEV   Hgt   Sex Smoke
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1   301     9  1.71  57       0     0
2   451     8  1.72  67.5     0     0
3   501     7  1.72  54.5     0     0
4   642     9  1.56  53       1     0
5   901     9  1.90  57       1     0
6  1701     8  2.34  61       0     0
```

```r
stata <- read_stata("data/Stata/FEV.DAT.dta")
head(stata)
```

```
# A tibble: 6 x 6
     Id   Age   fev   Hgt   Sex Smoke
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1   301     9  1.71  57       0     0
2   451     8  1.72  67.5     0     0
3   501     7  1.72  54.5     0     0
4   642     9  1.56  53       1     0
5   901     9  1.90  57       1     0
6  1701     8  2.34  61       0     0
```

The `foreign` package included in Base-R can also be used to Reading and writing data stored by some versions of 'Epi Info', 'Minitab', 'S', 'SAS', 'SPSS', 'Stata', 'Systat', 'Weka',and for reading and writing some 'dBase' files.

RDS

```r
rds_example <- readRDS("data/RDS/BETACAR.DAT.rds")
head(rds_example)
```

```
# A tibble: 6 x 8
  `'Prepar'` `'Id'` `'Base1lvl'` `'Base2lvl'`
       <int>  <int>        <int>        <int>
1          1     71          298          116
2          1     73          124          146
3          1     80          176          200
4          1     83          116          180
5          1     90          152          142
6          1     92          106          106
# ... with 4 more variables: `'Wk6lvl'` <int>,
#   `'Wk8lvl'` <int>, `'Wk10lvl'` <int>,
#   `'Wk12lvl'` <int>
```

`rdata`

The `.rdata` format is R's specific format. Instead of using a `read.{something}` function, `.rdata` is read into the environment using `load(filename.rdata)` and retains the original name it had when it was last saved.

```
load("data/R/BETACAR.DAT.rdata")  #named betacar when it was last saved
head(betacar)
```

```
  Prepar Id Base1lvl Base2lvl Wk6lvl Wk8lvl Wk10lvl
1      1 71      298      116    174    178     218
2      1 73      124      146    294    278     244
3      1 80      176      200    276    286     308
4      1 83      116      180    164    238     308
5      1 90      152      142    290    300     270
6      1 92      106      106    246    206     304
  Wk12lvl
1     190
2     262
3     334
4     226
5     268
6     356
```

2 Descriptive Statistics

```
PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed, please make s
```

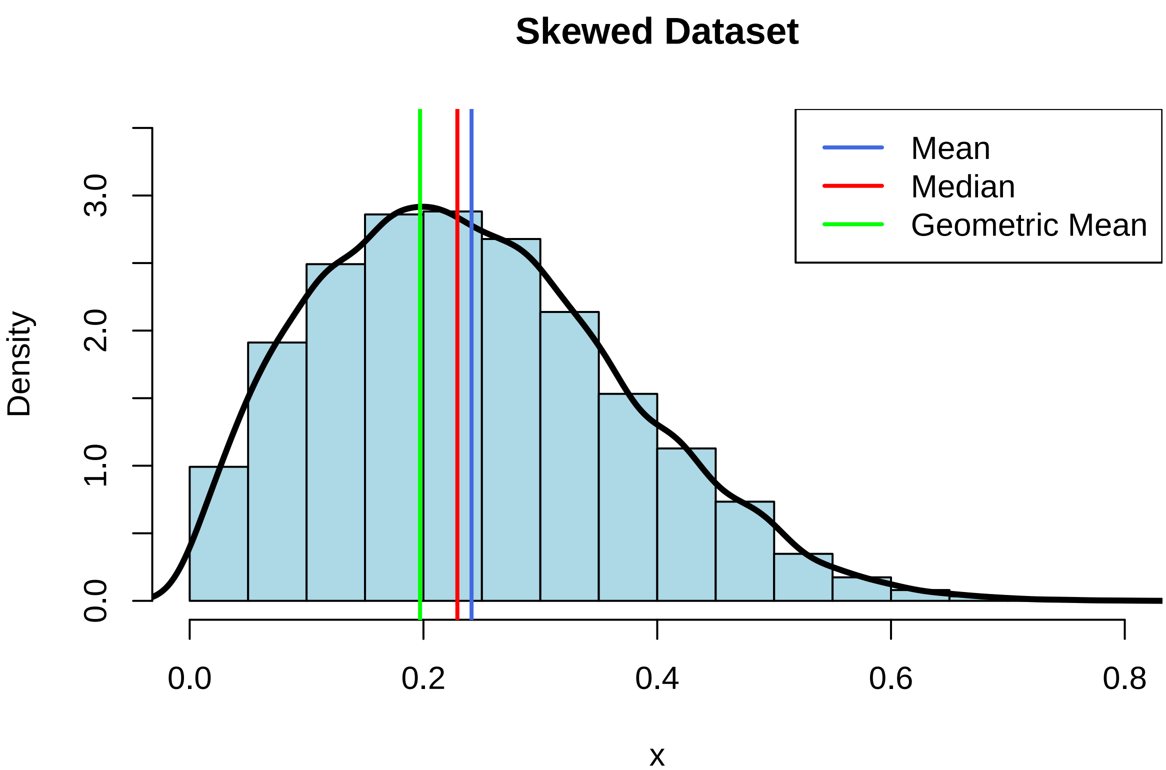
2.1 Measures of Location using Base R

Determining the correct method for measuring the central tendancy of a vector depends on the relationship between the numbers within the vector. Numbers that can be summed in a linear sequence are best represented using the arithmic mean.

If you're measuring units that add up as reciprocals in a sequence (such as speed or distance / time over a constant distance, capacitance in series, resistance in parallel), then a harmonic mean will give you a meaningful average. For example, the harmonic mean of capacitors in series represents the capacitance that a single capacitor would have if only one capacitor was used instead of the set of capacitors in series.

If you're measuring units that multiply in a sequence (such as growth rates or percentages), then a geometric mean will give you a meaningful average. For example, the geometric mean of a sequence of different annual interest rates over 10 years represents an interest rate that, if applied constantly for ten years, would produce the same amount growth in principal as the sequence of different annual interest rates over ten years did. Does an arithmetic mean of interest rates have any significance? As a number, sure. But as an "average" interest rate it seems less intuitive because the principal it produces at the end of ten years is much larger than the geometric mean. Similarly, the harmonic mean of interest rates produces a smaller principal, and so is less intuitive.

Now consider areas and volumes as a test of understanding. What mean should we use to report the "average" area or volume in a sequence of areas or volumes? Area is measured in units of length squared. Volume is measured in units of length cubed. In a sequence of areas or volumes, we could either add them up linearly and divide or multiply them and take the roots — which is correct? It depends on what we're measuring. If these areas or volumes are dependent upon each other (e.g., the size of the same microbe at different times), then a geometric mean probably makes more sense. If these areas or volumes are independent of each other (e.g., the size of a house or pool), then an arithmetic mean probably makes more sense. But whatever you decide, when in doubt report that decision. There is nothing worse for a reader than to see an "average" and not know how it was calculated! - Michael F. Martin,Quora Answer



2.1.1 The Arithmetic Mean

The arithmetic mean is the sum of all the observations divided by the number of observations. It is written in statistical terms as

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

```r
mean(ChickWeight$weight)
```

```
[1] 121.8
```

2.1.2 The Median

The sample median is:

1. If n is odd $\rightarrow \left(\frac{n+1}{2}\right)$th largest observation
2. If n is even $\rightarrow \left(\frac{n}{2}\right)$th and $\left(\frac{n}{2}+1\right)$th largest observations

```r
median(ChickWeight$weight)
```

```
[1] 103
```

2.1.3 The Mode

The mode is the most frequently occurring value among all observations in the sample. Although it is infrequently used, it is very useful for categorical and discrete data.

Since there isn't a built in R-function for mode, we learn how to write a function to return the mode through a few examples.

2.1.3.1 Functions

2.1.3.1.1 Base R Example

The most simple function begins by assigning the output of `function()` to some character string (e.g. `simple_fun`)

All statements after the `function()` are referred as the body of the function.

```r
function_name <- function(arg1, arg2,...) {
  #statements

  return("some output")
}
function_name() # returns NULL
```

```
[1] "some output"
```

Use `return()` to output the result of the function.

```
return_value <- function(x,y) {
  z=x-y
  z=x+y
  return(z)
}
return_value(4,5)
```

```
[1] 9
```

Since our goal is to find the most frequently occurring value in our data-set (`ChickWeight`), we need to decide the sequence of functions that we need to accomplish this. As you continue to add various R functions to your R tool belt, you will find many possible combinations for the same solution.

First, let's assign the weight column from ChickWeight to x to simplify things. When `x` is called, the weight column from ChickWeight is returned as a vector.

```
x<-ChickWeight$weight
head(x)
```

```
[1] 42 51 59 64 76 93
```

We can return the size of `x` using the `length` function. 578

```
length(x)
```

```
[1] 578
```

We can reduce x to return only the unique values by using the `unique` function. We'll assign it to y so we can use it later.

```
y <- unique(x)
length(y)
```

```
[1] 212
```

To more easily watch how the functions are working, we will create two data-frames to watch how we are manipulating both x and y.

```
df.x <- data.frame(x)
df.y <- data.frame(y)
```

Using the unique values from the `x` vector we defined as `y`, we can use the `match` function to return a vector that replaces each value in x with their position in the y vector (1-212).

```
df.x$position_in_y<-match(x, y)
head(df.x, n = 30)
```

```
    x position_in_y
1   42             1
2   51             2
```

```
3   59             3
4   64             4
5   76             5
6   93             6
7  106             7
8  125             8
9  149             9
10 171            10
11 199            11
12 205            12
13  40            13
14  49            14
15  58            15
16  72            16
17  84            17
18 103            18
19 122            19
20 138            20
21 162            21
22 187            22
23 209            23
24 215            24
25  43            25
26  39            26
27  55            27
28  67            28
29  84            17
30  99            29
```

The output from match can then be simplified using the tabulate function

```r
df.y$frequency <- tabulate(df.x$position_in_y)
head(df.y)
```

```
   y frequency
1 42        15
2 51         8
3 59         5
4 64         5
5 76         3
6 93         4
```

`which.max` returns the position of the maximum value.

```r
which.max(df.y$frequency)
```

```
[1] 43
```

```r
df.y[43,]  #df.y[row,column]
```

```
    y frequency
43 41        20
```

Putting it all together, we can do this in one line.

```r
df.y[which.max(tabulate(match(x,y))),]
```

```
    y frequency
43 41        20
```

```r
y[which.max(tabulate(match(x,y)))]
```

```
[1] 41
```

Writing this as a function

```r
mode <- function(x){
  unique_x <- unique(x)
  result<-unique_x[which.max(tabulate(match(x,unique_x)))]
  return(result)
}

mode(x)
```

```
[1] 41
```

2.1.3.1.2 Tidyverse Example

As with most problems in R, we can also find a solution using packages from the Tidyverse. We will therefore use this as an opportunity to introduce some of the basic tenants of Tidyverse functions.

In the `dplyr` package, a typical workflow will combine observations into a single data-frame, aggregate them into groups, manipulate values into new columns, and summaries the data-frame into more simple terms.

The piping operator `%>%` allows for this to be done seamlessly by literally pipping the result of one function into arguments of another function.

```r
print("non-piped text")
```

```
[1] "non-piped text"
```

```r
library(dplyr)
"piped text" %>% print()
```

```
[1] "piped text"
```

To show how this works, we will start with a simple example where we first want to divided the sum of three and some other number (e.g. 2) by seven.

Because of the order of operations, the sum of two and three would need to be placed with parenthesis to indicate it happens before dividing by seven.

```
(4+3)/7 # correct
```

```
[1] 1
```

```
4 + 3 / 7 # incorrect
```

```
[1] 4.429
```

The piping operator allows the order of operations be explicated dictated with manipulations of starting value reading from the left to right.

```
# pipes use the (.) as a placeholder
4 %>% + 3 %>% {./7} # removing the { } returns an error
```

```
[1] 1
```

Using pipes increases readability of your R-code and it can easily be reused for different starting values. In R Studio, the pipe character can be easily inserted using a keyboard shortcut (Windows:Ctrl+Shift+M, Mac:Cmd+Shift+M).

```
11 %>% + 3 %>% {./7}
```

```
[1] 2
```

Plus, the piped workflow can easily be defined by a function by assigning it to some string with a `.` in the beginning.

```
op_order <- . %>% +3 %>% {./7}
op_order(4)
```

```
[1] 1
```

```
op_order(11)
```

```
[1] 2
```

Determining Mode with `dplyr`

Using the `ChickWeight` data-set as before, we start by outlining the order of operations.

1. Group the data by weights `group_by()`
2. Tally the number of members within each group and sort by frequency. `tally()`
3. Select the row with the largest n. `slice()`
4. Return the corresponding weight. `.$weight`

```
ChickWeight %>% group_by(weight) %>% tally(sort = TRUE) %>% slice(1) %>% .$weight
```

```
[1] 41
```

As before, this workflow can be written as a function by placing `.` between the assignment operator `<-` and piping operator `%>%`.

```r
mode_cw<-. %>% group_by(weight) %>% tally(sort = TRUE) %>% slice(1) %>% .$weight

mode_cw(ChickWeight)
```

```
[1] 41
```

However, this function will only work on the `ChickWeight` data-set.

```r
mode_cw(mtcars)
```

```
Error in grouped_df_impl(data, unname(vars), drop): Column `weight` is unknown
```

2.1.4 Geometric Mean

The geometric mean is the antilogarithm of $\overline{\log x}$, where

$$\overline{\log x} = \frac{1}{n}\sum_{i=1}^{n}\log x_i$$

As with mode, there is no function in Base-R for finding the geometric mean.

```r
# using values
gm1 <- function(x){
  n = length(x)

  gm = exp((1/n)*sum(log(x)))

  return(gm)
}

gm2 <- function(x){
  return(exp(mean(log(x))))
}
```

```r
gm1(x)
```

```
[1] 103.1
```

```r
gm2(x)
```

```
[1] 103.1
```

2.2 Measures of Spread

2.2.1 Range

The range is the difference between the largest and smallest observations in a sample.

2.2.2 Quantiles/Percentiles

The pth percentile is defined by

1. The (k+1)th largest sample point if np/100 is not an integer (where k is the largest integer less than np/100).
2. The average of the (np/100)th and (np/100+1)th largest observations if np/100 is an integer.

```r
# 10th and 90th percentile
quantile(x = x, probs = c(0.1,0.9))
```

```
  10%   90%
 47.7 223.6
```

2.2.3 The Variance and Standard Deviation

$$s^2 = \frac{\sum_{i=1}^{n}(x-\bar{x})^2}{n-1}$$

```r
# variance
var(x)
```

```
[1] 5051
```

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x-\bar{x})^2}{n-1}}$$

```r
# Standard deviation
sd(x)
```

```
[1] 71.07
```

2.3 The Coefficient of Variation

The coefficient of variation (CV) is defined by

$$100\% \times \frac{s}{\bar{x}}$$

2.4 Grouped data

```r
bwt <- readr::read_csv("data/CSV/Birthweight.csv")
bwt
```

```
# A tibble: 100 x 1
     BWT
   <int>
 1    58
 2   120
 3   123
 4   104
 5   121
 6   111
 7    91
 8   104
 9   128
10   133
# ... with 90 more rows
```

Frequency Distribution

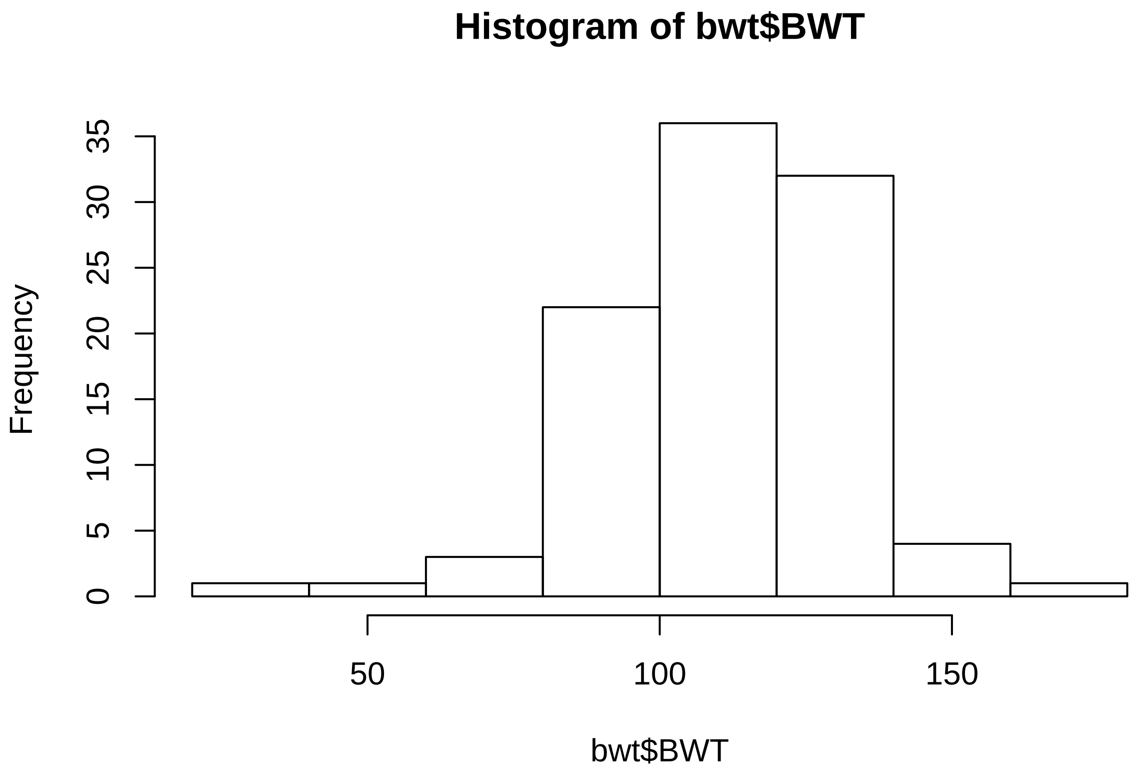
```r
# starting dataframe (df)
bwt %>%
  # sort df by BWT column
  arrange(BWT) %>%
   # counts values in BWT (n)
  add_count(BWT) %>%
  # renames n to Frequency
  rename(Frequency = n) %>%
  # creating new columns
  mutate(
    Cum_Percent = cume_dist(BWT)      # returns cumulative percent
  ) %>%
  # remove duplicated rows
  distinct(.) -> freq_tab

DT::datatable(freq_tab)
```

2.5 Graphic Methods

2.5.1 Bar Graphs

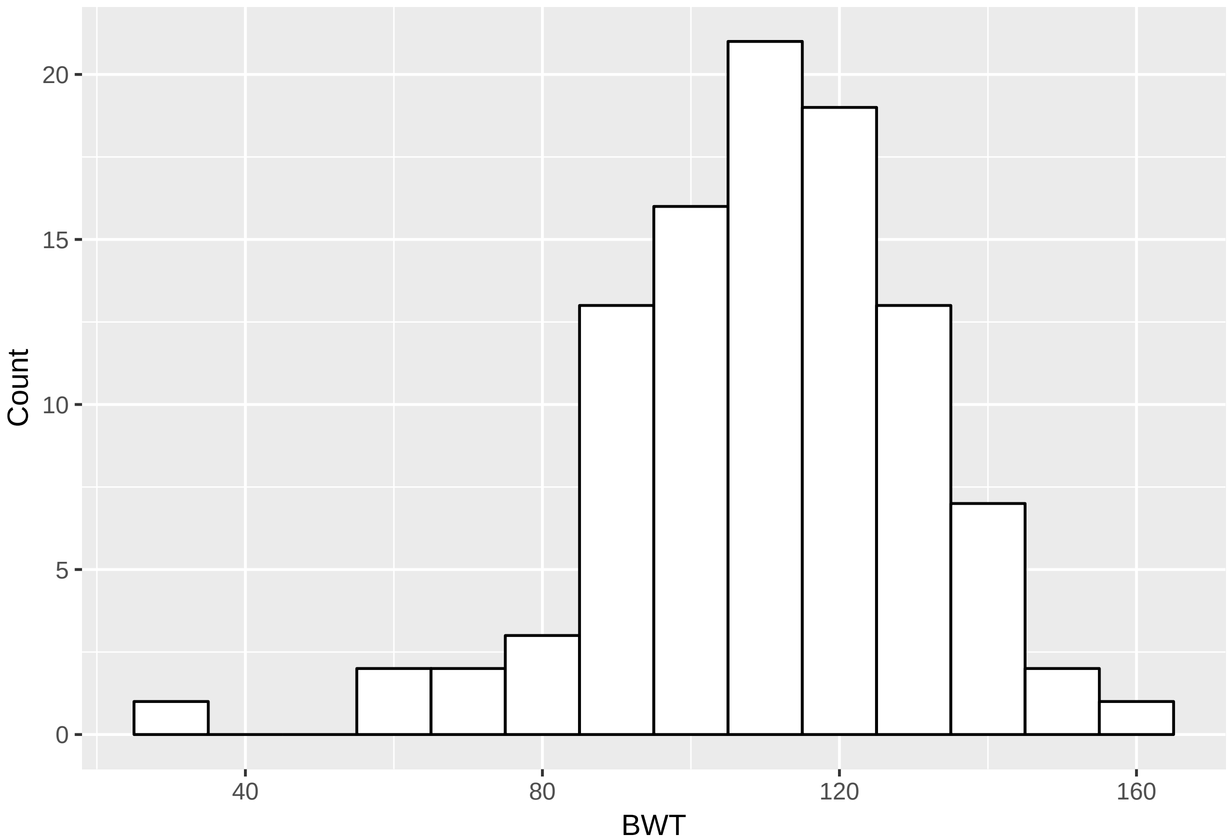
Base-R

```r
hist(bwt$BWT)
```



ggplot2

```r
library(ggplot2)
 ggplot(data = bwt,aes(BWT))+
  geom_histogram(fill = "white", color = "black",binwidth = 10)+
  ylab("Count")
```



2.5.2 Stem-and-Leaf Plots

Base-R

```
stem(bwt$BWT, scale = 2)
```

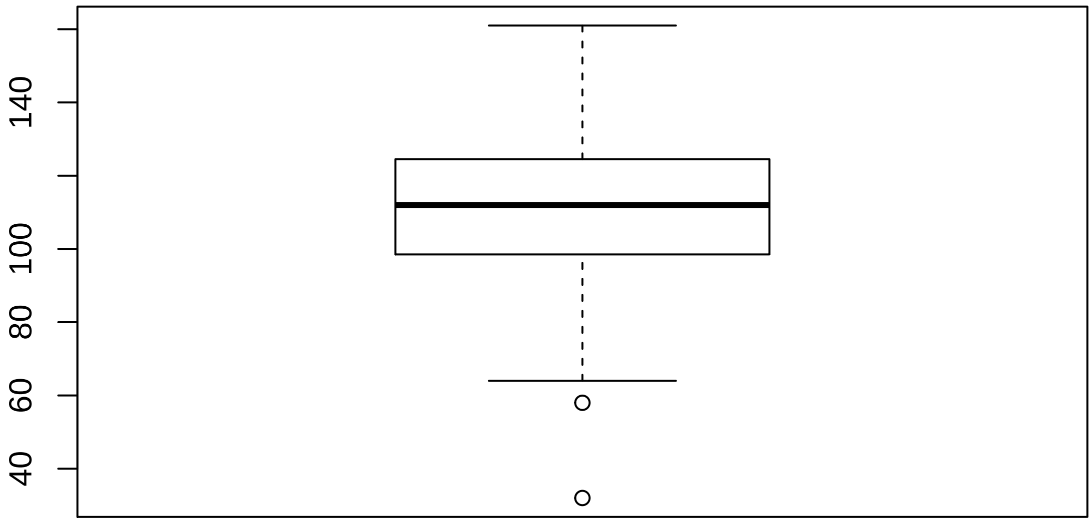
```
  The decimal point is 1 digit(s) to the right of the |

   3 | 2
   4 | 
   5 | 8
   6 | 478
   7 | 
   8 | 3556788999
   9 | 12344568889
  10 | 0123444445567888899
  11 | 00122235555556889
  12 | 01112222344445567788
  13 | 222334557888
  14 | 0146
  15 | 5
```

```
16 | 1
```

2.5.3 Box Plots

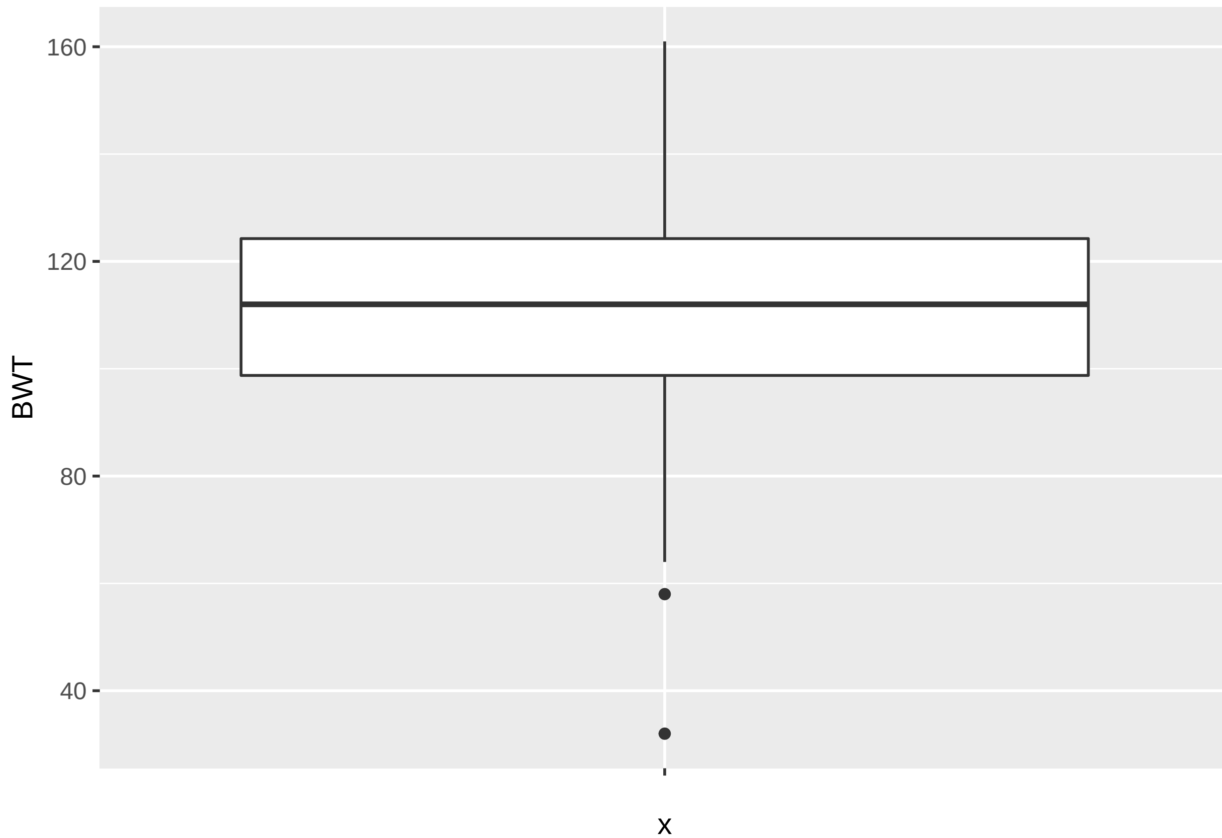
Base-R

```r
boxplot(bwt$BWT)
```



ggplot2

```r
ggplot(bwt, aes(x = "",BWT))+geom_boxplot()
```


160
120
80
40
BWT
x

3 Probability

3.1 Introduction

3.2 Definition of Probability

3.3 Some Useful Probabilistic notation

3.4 The Multiplication Law of Probability

3.5 The Addition Law of Probability

3.6 Conditional Probability

3.7 Bayes' Rule and Screening Tests

3.8 Bayesian inference

3.9 RoC Curves

3.10 Prevalence and incidence

4 Discrete Probability distributions

4.1 Introduction

4.2 Random Variables

4.3 The Probability-Mass Function for a Discrete Random Variable

4.4 The Expected Value of a discrete Random Variable

4.5 The Variance of a Discrete Random Variable

4.6 The Cumulative-Distribution Function of a Discrete Random Variable

4.7 Permutations and Combinations

4.8 The Binomial distribution

4.9 Expected Value and Variance of the Binomial distribution