SML201 Chapter 2.3

EDA: Part II

Daisy Huang

Fall 2019

Contents

R functions covered

- Functions/techniques that help you to get familiar with the dataset: `nrow()`, `ncol()`'
- Create a function: `function(input.variable1 = , input.variable2 = ,...){...}`

- Statistic functions: `tapply(X = , INDEX = , FUN = )`, `cor(x = , y = )`
- Data manipulation: `merge(x = , y = , by = , by.x = , by.y = , all = , all.x = , all.y = )`, `mutate()` in `dplyr` package,
- Graphical summary functions: `boxplot(y ~ grp, xlab = , ylab = , main = , names = , ...)`, plot(x = , y = , xlab = , ylab = , main = ,...)
- Graphical options: `par(mfrow = c(..., ...))`, `boxplot(..., ylab = ..., main = ...)`

Introduction

In the last chapter we talked about how to manipulate vectors and data frames in R and we also learnt various statistics that are useful to extract summary information from a dataset. With these tools we were able to answer a lot of interesting questions about the Titanic passengers by manipulating the columns of the Titanic dataset one at a time. However, more (interesting) questions can be answered if we can use multiple columns of the dataset.

Read in the dataset

```
t.ship =
  read.csv(file =
  '/Users/billhaarlow/Desktop/SML201/titanic.csv')
```

We should refresh our memory on the properties of the dataset.

```
class(t.ship)  # this tells me what kind of R object this dataset is
[1] "data.frame"
dim(t.ship)  # 891 rows by 12 columns
[1] 891  12
```

```
head(t.ship)  # look at the first 6 (by default) rows of an object
  PassengerId Survived Pclass
1           1        0      3
2           2        1      1
3           3        1      3
4           4        1      1
5           5        0      3
6           6        0      3
                                                 Name    Sex
1                             Braund, Mr. Owen Harris   male
2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) female
3                              Heikkinen, Miss. Laina female
4        Futrelle, Mrs. Jacques Heath (Lily May Peel) female
5                            Allen, Mr. William Henry   male
6                                    Moran, Mr. James   male
```

```
  Age SibSp Parch           Ticket    Fare Cabin Embarked
1  22     1     0        A/5 21171  7.2500             S
2  38     1     0         PC 17599 71.2833   C85       C
3  26     0     0 STON/O2. 3101282  7.9250             S
4  35     1     0           113803 53.1000  C123       S
5  35     0     0           373450  8.0500             S
6  NA     0     0           330877  8.4583             Q
```

```
str(t.ship, strict.width = "cut")
'data.frame':	891 obs. of  12 variables:
 $ PassengerId: int  1 2 3 4 5 6 7 8 9 10 ...
 $ Survived   : int  0 1 1 1 0 0 0 0 1 1 ...
 $ Pclass     : int  3 1 3 1 3 3 1 3 3 2 ...
 $ Name       : Factor w/ 891 levels "Abbing, Mr. Anthony",....
 $ Sex        : Factor w/ 2 levels "female","male": 2 1 1 1 2..
 $ Age        : num  22 38 26 35 35 NA 54 2 27 14 ...
 $ SibSp      : int  1 1 0 1 0 0 0 3 0 1 ...
 $ Parch      : int  0 0 0 0 0 0 0 1 2 0 ...
 $ Ticket     : Factor w/ 681 levels "110152","110413",..: 52..
 $ Fare       : num  7.25 71.28 7.92 53.1 8.05 ...
 $ Cabin      : Factor w/ 148 levels "","A10","A14",..: 1 83 ..
 $ Embarked   : Factor w/ 4 levels "","C","Q","S": 4 2 4 4 4 ..
```

Will class or gender affect the survival rate? How big are the gaps between the socioeconomic classes?

- What is the survival rate for each ticket class? Are the rates about the same across the three ticket classes?

```
# number of survivals in each ticket class
numerator = table(t.ship$Pclass, t.ship$Survived)
# this produces a contingency table
dim(numerator)
[1] 3 2
class(numerator)
[1] "table"

# number of people in each ticket class
denominator = table(t.ship$Pclass)  # 1-dimensional array
dim(denominator)
[1] 3
class(denominator)
[1] "table"

# The following will not work since array operations require
# the two arrays to be in the same dimension
# numerator/denominator

# however, you can divide an array by a vector if the
# dimensions make sense
table(t.ship$Pclass, t.ship$Survived)/as.vector(table(t.ship$Pclass))

             0         1
  1 0.3703704 0.6296296
  2 0.5271739 0.4728261
  3 0.7576375 0.2423625


# In short, you cannot operate on two arrays unless both
# arrays have exactly the same dimensions; e.g.,
# table(t.ship$Pclass, t.ship$Survived)/table(t.ship$Pclass)
```

As we can see above, the procedure gets complicated if we want to rely on using the function `table()` only. Fortunately, there is a function `tapply()` that is suitable for this kind of tasks. Here is how we would use the function `tapply()` to accomplish the same task. Note that you can obtain the answer with only ONE line of code!

```
tapply(X = t.ship$Survived, INDEX = t.ship$Pclass, FUN = mean)
        1         2         3
0.6296296 0.4728261 0.2423625
```

Exercise

Question 1. What is the survival rate for each sex? Are the rates about the same?

```
tapply(t.ship$Survived, INDEX = t.ship$Sex, FUN = mean)
   female      male
0.7420382 0.1889081
```

The survival rate is a lot higher for women.

Question 2. What is the average and median price for each ticket class?

```
# average prices
tapply(t.ship$Fare, INDEX = t.ship$Pclass, FUN = mean)
       1        2        3
84.15469 20.66218 13.67555

# how big are the gaps?
avg.tic.price = tapply(t.ship$Fare, INDEX = t.ship$Pclass, FUN = mean)
(avg.tic.price[1:2] - avg.tic.price[2:3])/avg.tic.price[2:3]
       1        2
3.072885 0.510885
# 1st class tickets are about 3 times more than the 2nd class
# tickets on average 2st class tickets are about 150% of the
# 2nd class tickets on average


# median prices
tapply(t.ship$Fare, INDEX = t.ship$Pclass, FUN = median)
      1       2       3
60.2875 14.2500  8.0500

med.tic.price = tapply(t.ship$Fare, INDEX = t.ship$Pclass, FUN = median)
(med.tic.price[1:2] - med.tic.price[2:3])/med.tic.price[2:3]
        1         2
3.2307018 0.7701863
# 1st class tickets are about 3 times more than the 2nd class
# tickets in terms of the median 2st class tickets are almost
# 180% of the 2nd class tickets in terms of the median
```

Multiple histograms

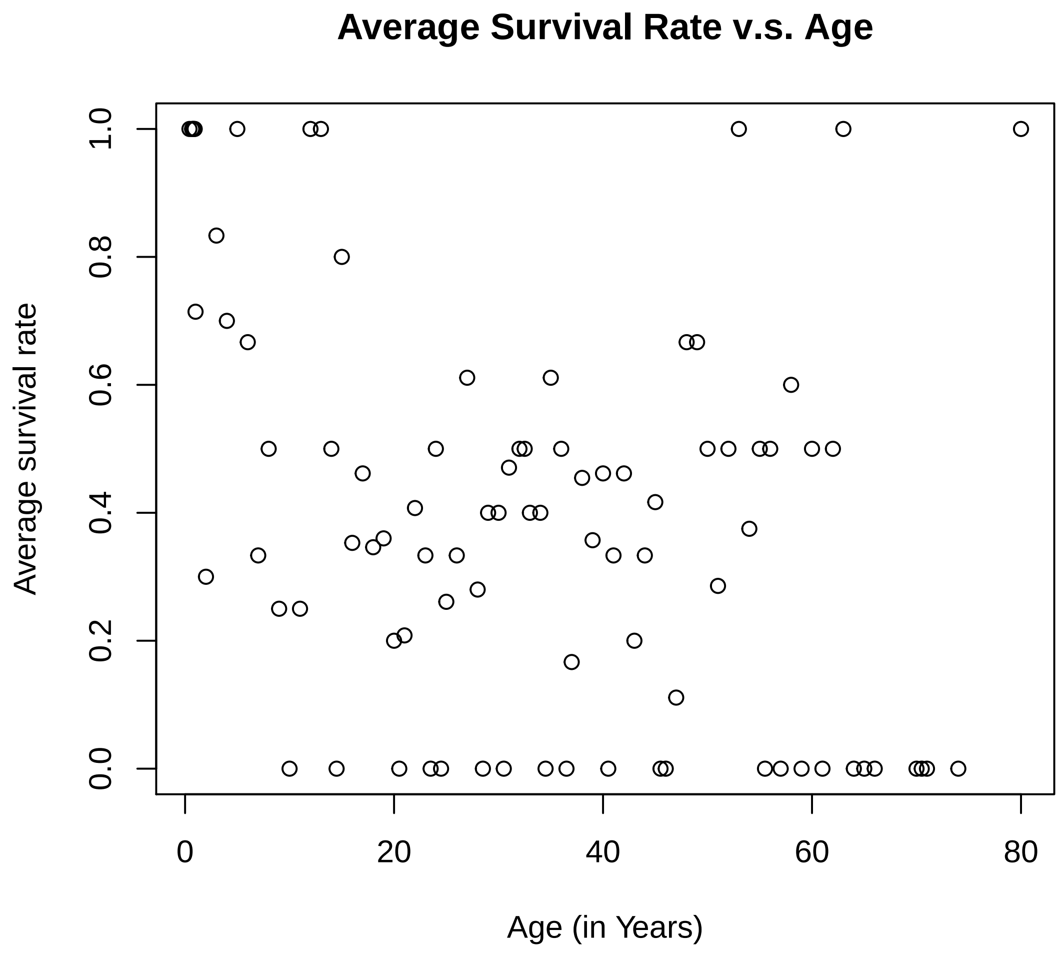
Does age affect the survival rate?

A natural way to approach this question would be to see if there is a relationship between `Age` and the average survival rate for people of that age.

```
# average survival rate for each age
s.rate.by.age = tapply(X = t.ship$Survived, INDEX = t.ship$Age,
    FUN = mean)
```

We can use a scatterplot to investigate this relationship.

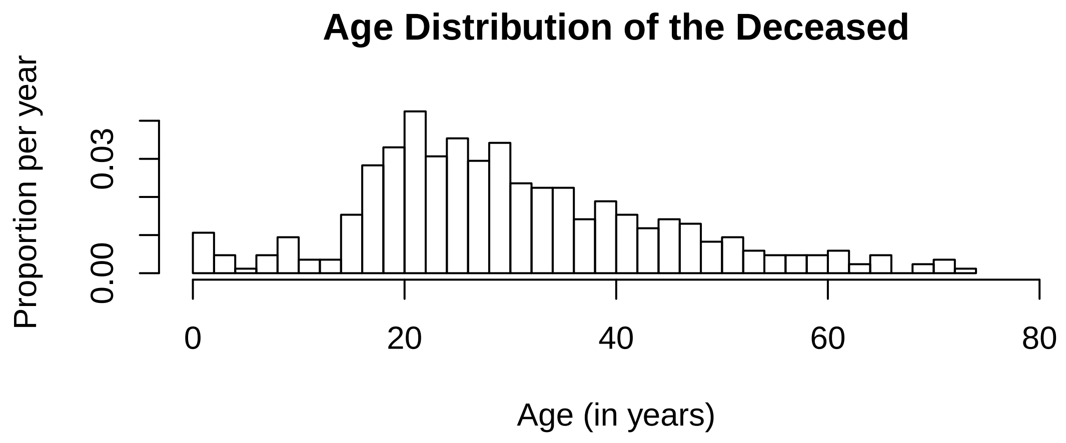
```
plot(x = as.numeric(names(s.rate.by.age)), y = s.rate.by.age,
    main = "Average Survival Rate v.s. Age", xlab = "Age (in Years)",
    ylab = "Average survival rate")
```

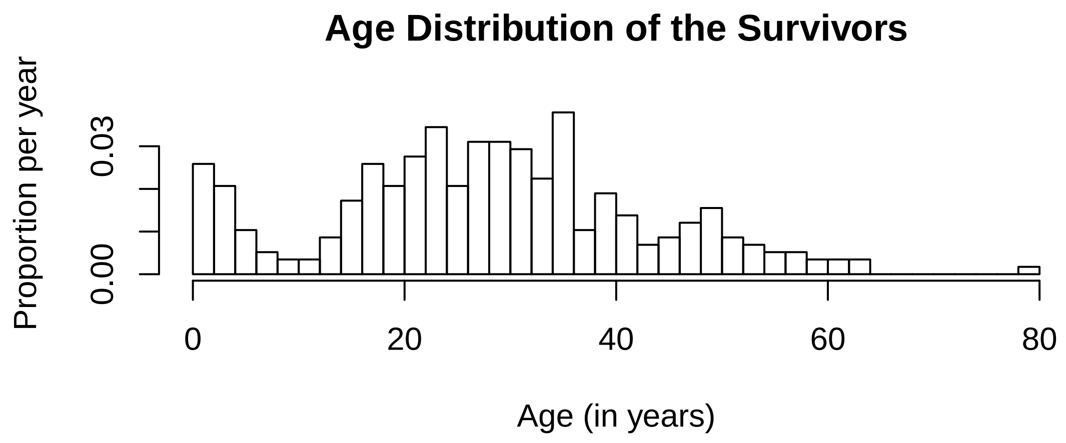


Note that this does not tell us how many people we have in each age group; thus, it is difficult to say the 100% survival rate was the result due to having too few people in the age group or not. We can try looking at the problem from a slightly different angle by answering the following question.

Question 3. Can we compare the age distributions of the passengers between the survivors and the deceased?

```r
# histograms
par(mfrow = c(2, 1))
hist(t.ship$Age[t.ship$Survived == 0], freq = F, breaks = 30,
    main = "Age Distribution of the Deceased", xlab = "Age (in years)",
    ylab = "Proportion per year", xlim = c(0, 80))
hist(t.ship$Age[t.ship$Survived == 1], freq = F, breaks = 30,
    main = "Age Distribution of the Survivors", xlab = "Age (in years)",
    ylab = "Proportion per year", xlim = c(0, 80))
```





Side-by-side boxplots

Previously, we learnt how to interpret a boxplot and how to draw a boxplot for one variable. We also learnt that a boxplot is a graphical way to display the five number summaries. Because of this a boxplot does not give us as much detailed information about the distribution of the data as a histogram; however, when comparing multiple datasets using side-by-side boxplots makes it convenient to detect any shifts in the distributions.

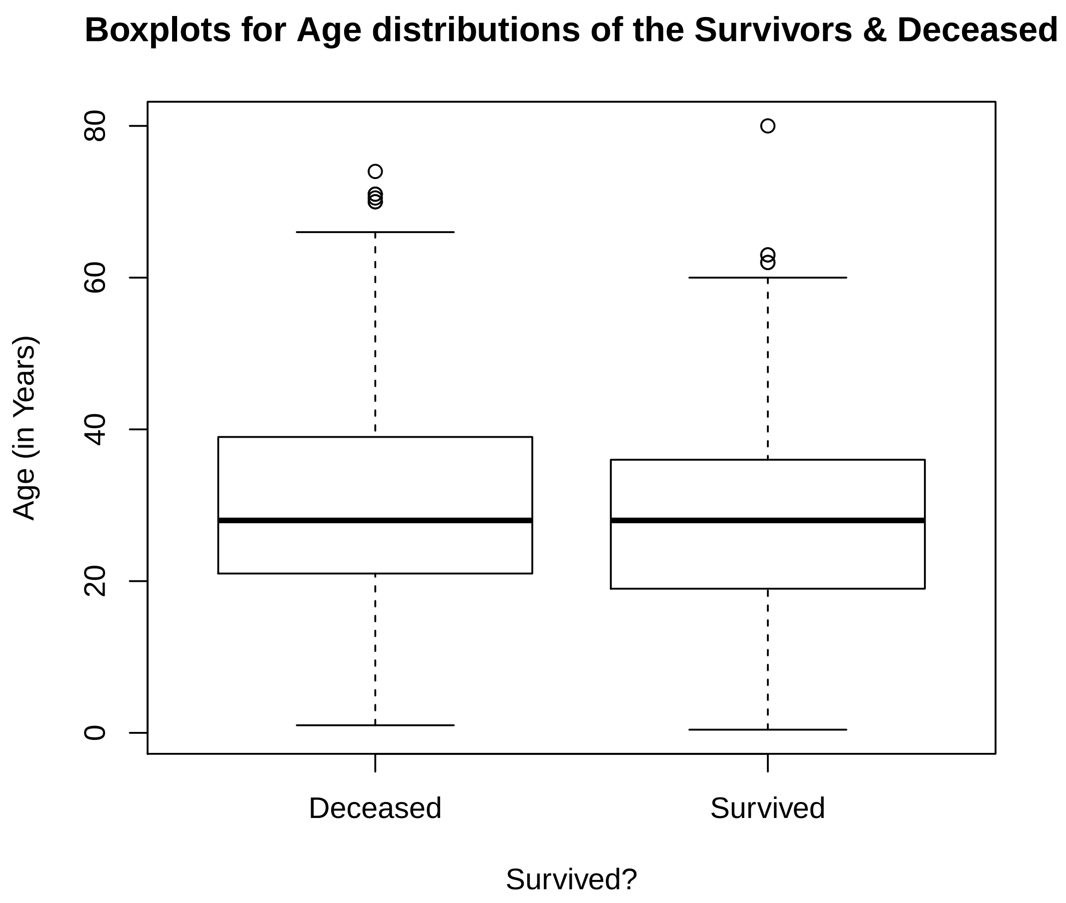
To make side-by-side boxplots the `boxplot()` function takes its input arguments in this form

boxplot(y ~ grp)

where `grp` will be converted into a factor (a categorical variable) if not already so; `R` uses `grp` to divide your dataset into different groups/subsets–think about what the `INDEX` argument in `tapply()` does.

```
# boxplots
```

```r
par(mfrow = c(1, 1))
boxplot(t.ship$Age ~ t.ship$Survived, names = c("Deceased", "Survived"),
    main = "Boxplots for Age distributions of the Survivors & Deceased",
    xlab = "Survived?", ylab = "Age (in Years)")
```



Creating functions

User-defined functions are often used to reduce the number of lines of code needed when one wants to repeat a set of task many times on different R objects. This makes the code easier to read and reduces the chance of having errors in the code compared to using the copy-and-paste method.

Question 4. Can we create a function to calculate the survival rate and median age when users specify the gender and the ticket class of the group of passengers?

```r
InfoOnGenderPclass = function(gender, ticket.class) {
    # `gender` options: 'female' and 'male'; `ticket.class`
    # options: 1-3
    subs.titanic = t.ship[t.ship$Sex == gender & t.ship$Pclass ==
        ticket.class, ]
    survival.rate = mean(subs.titanic$Survived)
    return(c(survival.rate = mean(subs.titanic$Survived), median.age = median(subs.titanic$Age,
        na.rm = T)))
}

# run to check the output of the function
InfoOnGenderPclass(gender = "female", ticket.class = 2)
survival.rate    median.age
    0.9210526    28.0000000
InfoOnGenderPclass(gender = "male", ticket.class = 3)
survival.rate    median.age
    0.1354467    25.0000000

# assign output to a new variable
male3.s.rate.med.age = InfoOnGenderPclass(gender = "male", ticket.class = 3)
```

Some interesting facts about this dataset

Someone asked in a previous lecture why the numbers for the variable `Fare` have 4 digits after the decimal. I did some research and found out that the ticket prices were in the format of Lsd; i.e., they are in the format of Pounds-Shillings-Pence. (https://www.encyclopedia-titanica.org/titanic-survivor/annie-moore-ward.html)

Note that

- 1 pound = 240 pence
- 1 shilling = 12 pence
- 1 pound = 20 shillings

Thus, in 1912 the British currency was not in the metric system (https://en.wikipedia.org/wiki/%C2%A3sd).

Also, there are discussions online about what exactly the definition for the variable `Fare` should be and there seems to be a lot of confusions. (You can see some of the discussions here https://www.kaggle.com/c/titanic/discussion/3916)

After reading a few of the webpages for Titanic passengers I was able find out the definition for `Fare`. For example, on the webpage for the passenger Ann Moore Ward (https://www.encyclopedia-titanica.org/titanic-survivor/annie-moore-ward.html) it says

> For the return to the USA Annie boarded the Titanic... with Mrs Cardeza, her son Thomas and his manservant Gustave Lesueur and the entourage travelled on **ticket number 17755** which cost the princely **sum** of £512, 6s, 7d.

Thus, the figures shown for the variable `Fare` are not for individual passengers but for the group of passengers sharing the same ticket number!

We will figure out the single ticket prices ourselves.

Let's create a new data frame that includes a column for the single ticket prices.

First, let's find out how many people there are for each ticket number.

```
# the output for table() is a 1-dimensional array however, it
# is much easier to work with vectors; thus, we want to
# convert the output into a vector; note that `as.vector()`
# will remove the names of the elements in the table by
# default.

# here we want to find out the number of passengers for each
# ticket number
no.grp.members = as.vector(table(t.ship$Ticket))
head(no.grp.members)
[1] 3 3 2 1 1 1
head(table(t.ship$Ticket))

110152 110413 110465 110564 110813 111240
     3      3      2      1      1      1
```

Note that the length of `no.grp.members` is smaller than the number of rows in `t.ship`. Is this expected?

```
# How many unque ticket numbers there are?
length(no.grp.members)
[1] 681

# How many passengers?
nrow(t.ship)
[1] 891
```

Merging datasets

We will use the function `merge()`. Let's create a data frame for ticket numbers and group member sizes for merging.

```
ticket.member = data.frame(Ticket = names(table(t.ship$Ticket)),
    grp.member.ct = no.grp.members)
head(ticket.member)
  Ticket grp.member.ct
1 110152             3
2 110413             3
3 110465             2
4 110564             1
5 110813             1
6 111240             1
```

```
# `by.x` is the column in `x` that is used for the matching
# `by.y` is the column in `y` that is used for the matching
```

```
t.ship.merged = merge(x = t.ship, y = ticket.member, by.x = "Ticket",
    by.y = "Ticket")

# Check that no one is missing
dim(t.ship.merged)
[1] 891  13
dim(t.ship)
[1] 891  12
```

Adding additional columns to a data frame

```
# load the package `dplyr` so that we can use the function
# `mutate()`
library(dplyr)

Attaching package: 'dplyr'
The following objects are masked from 'package:stats':

    filter, lag
The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union

t.ship.complete = mutate(t.ship.merged, single.tic.price = Fare/grp.member.ct)

knitr::kable(head(t.ship.complete))
```

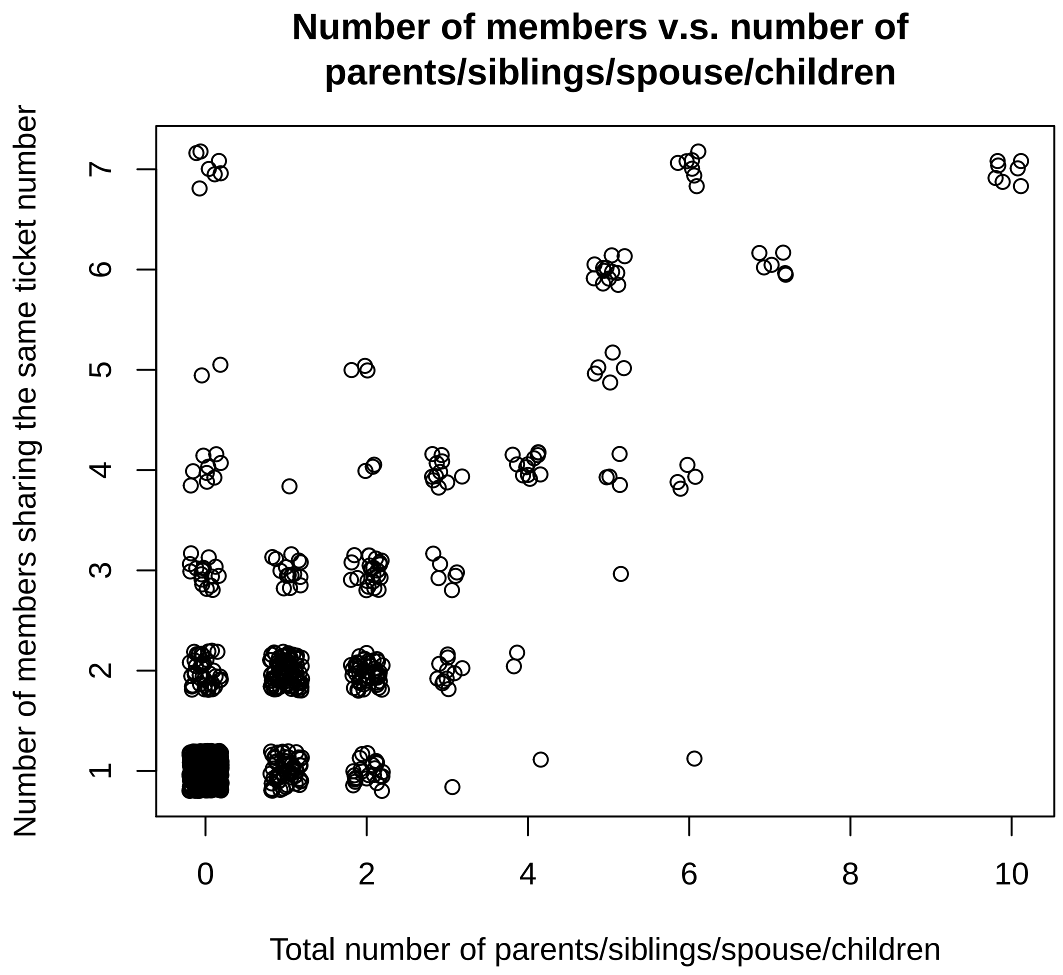
Ticket	PassengerId	Survived	Pclass	Name	Sex
110152	258	1	1	Cherry, Miss. Gladys	female
110152	760	1	1	Rothes, the Countess. of (Lucy Noel Martha Dyer-Edwards)	female
110152	505	1	1	Maioni, Miss. Roberta	female
110413	559	1	1	Taussig, Mrs. Emil (Tillie Mandelbaum)	female
110413	263	0	1	Taussig, Mr. Emil	male
110413	586	1	1	Taussig, Miss. Ruth	female

Checking the relationship between two variables

Scatterplot

We have seen how we can investigate the relationship between two variables by using a scatterplot. Here we would like to see if there is a relationship between the number of members sharing the same ticket number and the number of parent/sibling/spouse/child companions.

```
plot(x = jitter(t.ship.complete$SibSp + t.ship.complete$Parch),
    y = jitter(t.ship.complete$grp.member.ct), xlab = "Total number of parents/siblings/spouse/cl
    ylab = "Number of members sharing the same ticket number",
    main = "Number of members v.s. number of \n parents/siblings/spouse/children")
```



Note that the trend of the relationship between the `x` and `y` variables is somewhat linear.

Checking the *linear* relationship between two variables

A Correlation coefficient is a numerical measure of the strength of the **linear** relationship between two variables. We will talk about this in more details in lecture.

A correlation coefficient is always between -1 and 1. A positive value means a directly proportional linear relationship. A negative correlation implies an indirectly proportional

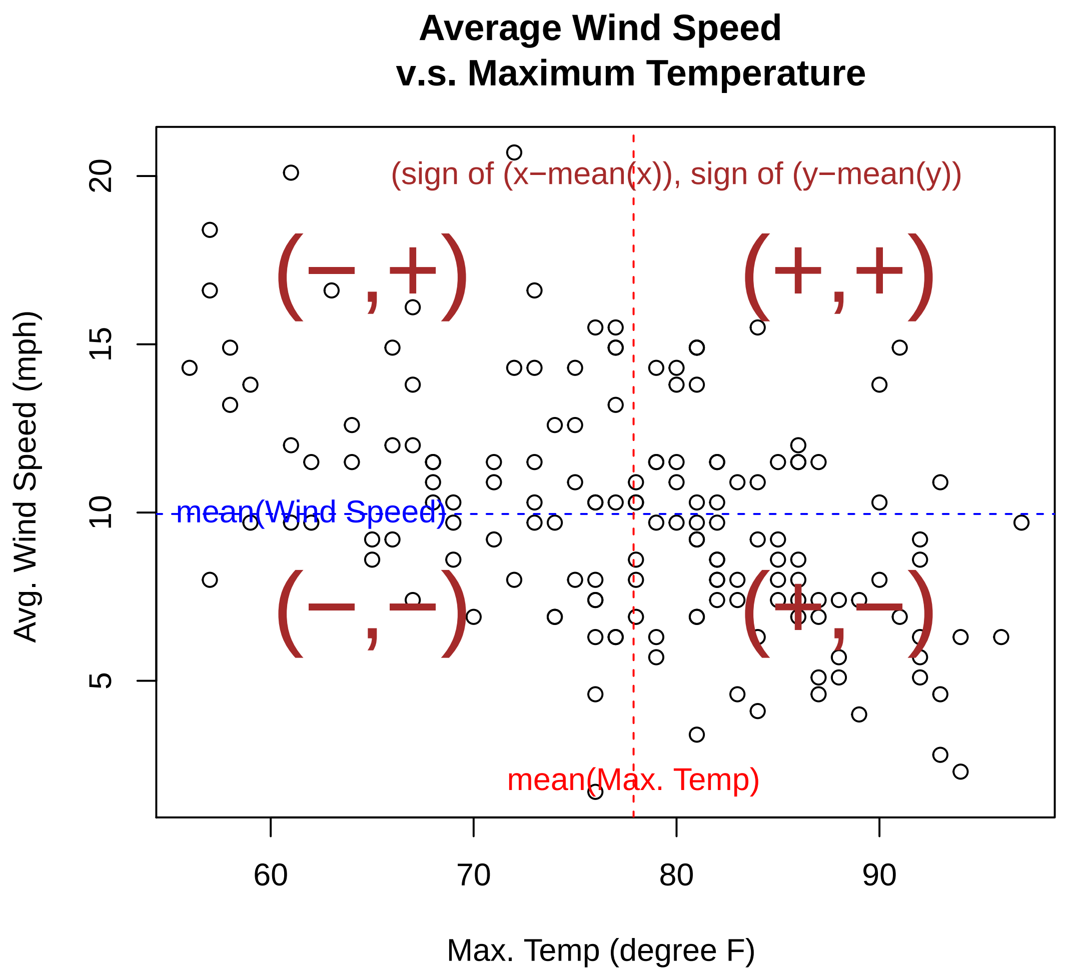


Figure 1: Scatterplot for Avg. Wind Speed v.s. Max. Temp.(May 1-Sept 30, 1973)

relationship. A value close to zero implies no linear association. A value close to 1 or -1 implies strong linear relationship.

As an example here let's look at the relationship between the average wind speed and the maximum temperature of the day for days between May 1 and Sept 30 in 1973.

For the two variables that we plotted above the correlation between `Wind Speed` and the `Maximum Temperature` is -0.46.

Note that both cor(x, y) and cor(y, x) give you the same value.

```
cor(t.ship.complete$SibSp + t.ship.complete$Parch, t.ship.complete$grp.member.ct)
[1] 0.7484868
```

However, no *linear* association is not the same as no association; for example,

```
# <center>![Correlation](/Users/daisyhuang/Documents/Teaching/SML201_Fall2018/Lectures/EDA/EDA_p
```

https://en.wikipedia.org/wiki/Correlation_and_dependence

Technical details related to concepts and functions covered in this chapter

We learnt in last chapter the different data types in R: integer, numeric, logical, character/string, factor and complex.

We also learnt two object types in R: vector and data frame. Recall that a data frame can have entries with different data types whereas a vector has to have homogeneous data type. It turns out that vectors and data frames are from different groups of objects in R.

Two main classes of objects in R:

- With homogeneous elements: Vectors, matrices, arrays.
- With heterogeneous elements: Lists, data frames.

Examples of different R objects

Vector

Please see previous chapter for examples of vectors.

Matrix– a "vector" with 2-dimensional shape

```
x2 = matrix(1:6, nrow = 3, ncol = 2)
class(x2)
[1] "matrix"
is.vector(x2)
[1] FALSE
# Prefix `is.` to any object type name will give you a
# function that checks whether the input is of the particular
# data type
is.data.frame(x2)
[1] FALSE
is.matrix(x2)
[1] TRUE
dim(x2)
[1] 3 2
length(x2)
[1] 6
```

Array–a "vector" with 3 or more dimensional shape

```
x3 = array(1:24, c(2, 3, 4))
x3
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18

, , 4

     [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24
# 4 blocks of matrices, each of size 2x3.
```

List–can contains elements of *different* data types

```
mylist = list(elt1 = 3:9, elt2 = c(2.3, 1.5), elt3 = c("a", "zz"),
    elt4 = matrix(1:4, ncol = 2))
mylist  # you can have whatever you like for the elements on a list
$elt1
[1] 3 4 5 6 7 8 9

$elt2
[1] 2.3 1.5

$elt3
[1] "a"  "zz"

$elt4
     [,1] [,2]
[1,]    1    3
[2,]    2    4
class(mylist)
[1] "list"
```

Data.frame–a special type of list which contains variables of the same number of rows

```
`?`(data.frame  # see the definition of a data.frame in the 'details' section
)

dat = data.frame(ind = 1:4, temperature = c(73.2, 75.6, 81.1,
    89), greater.than.86 = (c(73.2, 75.6, 81.1, 89) > 86), hot = c("No",
    "No", "No", "Yes"))
dat
  ind temperature greater.than.86 hot
1   1        73.2           FALSE  No
2   2        75.6           FALSE  No
3   3        81.1           FALSE  No
4   4        89.0            TRUE Yes
class(dat)
[1] "data.frame"
```

Often time data frames can be manipulated in similar ways as for matrices and lists. However, it is important to understand that data frame is a very different object type from matrix.

Changing the data type of a vector

Note that for the three R objects above their elements always have the same data type. You can use the functions `as._data type_()` to change the data type of the elements in these R objects. Why do we want to change the data type of the elements in a R object? Having the appropriate data type allows R to extract the relevant information; e.g.,

```
str(t.ship, strict.width = "cut")
'data.frame':   891 obs. of  12 variables:
 $ PassengerId: int  1 2 3 4 5 6 7 8 9 10 ...
 $ Survived   : int  0 1 1 1 0 0 0 0 1 1 ...
 $ Pclass     : int  3 1 3 1 3 3 1 3 3 2 ...
 $ Name       : Factor w/ 891 levels "Abbing, Mr. Anthony",....
 $ Sex        : Factor w/ 2 levels "female","male": 2 1 1 1 2..
 $ Age        : num  22 38 26 35 35 NA 54 2 27 14 ...
 $ SibSp      : int  1 1 0 1 0 0 0 3 0 1 ...
 $ Parch      : int  0 0 0 0 0 0 0 1 2 0 ...
 $ Ticket     : Factor w/ 681 levels "110152","110413",..: 52..
 $ Fare       : num  7.25 71.28 7.92 53.1 8.05 ...
 $ Cabin      : Factor w/ 148 levels "","A10","A14",..: 1 83 ..
 $ Embarked   : Factor w/ 4 levels "","C","Q","S": 4 2 4 4 4 ..
# currently `t.ship$Pclass` is an integer vector; thus when
# we apply the `summary()` function to `t.ship$Pclass` we
# have

summary(t.ship$Pclass)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   2.000   3.000   2.309   3.000   3.000
```

```
# however, if we tell R that the elements in `t.ship$Pclass`
# actually represent categories we have

summary(as.factor(t.ship$Pclass))
  1   2   3 
216 184 491 
# which data type is more appropriate here?
```

We already talked about construction, concatenation, extraction and assignment for vectors and data frames; here we are going to talk about these techniques for other R objects.

Construction, concatenation, extraction and assignment for some R objects

Matrices

You can create a matrix in R using the matrix function.

```
m = matrix(1:6, nrow = 2, ncol = 3)
# By default matrices are fille by column in R.
m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6


m <- matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
# You can change the assignment order by rows.
m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

You can **assign** names to the rows and columns

```
rownames(m) = c("r1", "r2")
colnames(m) = c("c1", "c2", "c3")
m
   c1 c2 c3
r1  1  2  3
r2  4  5  6
```

To **index** elements of a matrix, use the same **five methods of indexing that we have for vectors** but with the first index for rows and the second for columns.

```
m[-1, 2]  # Exclusion and inclusion by position
[1] 5
m["r1", ]  # By name, also empty index means 'all'
c1 c2 c3 
 1  2  3 
m["r1", c("c1", "c2")]
```

```
c1 c2
 1  2
m[, c(T, T, F)]
   c1 c2
r1  1  2
r2  4  5
# Empty row index means taking all the rows logical column
# index (note: you will need to have the same number of index
# as the number of columns)

# What happens if we run
m[, c(T, F)]
   c1 c3
r1  1  3
r2  4  6
m
   c1 c2 c3
r1  1  2  3
r2  4  5  6
```

You can **combine** multiple matrices together

```
m
   c1 c2 c3
r1  1  2  3
r2  4  5  6
m2 = matrix(11:16, ncol = 3)
m2
     [,1] [,2] [,3]
[1,]   11   13   15
[2,]   12   14   16

cbind(m, m2)  # combine 2 matrices side by side by columns
   c1 c2 c3
r1  1  2  3 11 13 15
r2  4  5  6 12 14 16

rbind(m, m2)  # stack 2 matrices by rows
   c1 c2 c3
r1  1  2  3
r2  4  5  6
   11 13 15
   12 14 16
```

Aside: To get more info on your matrix you can use these functions that you already leant for data frames:

```
dim(m)  # The size of m
[1] 2 3
```

```
nrow(m)  # The number of rows in m
[1] 2
ncol(m)  # The number of columns in m
[1] 3
```

The functions above also can be used for data frames.

You can also take the transpose of your matrix this way

```
t(m)  # Transpose of m
   r1 r2
c1  1  4
c2  2  5
c3  3  6
```

If you apply `t()` to a data frame `R` will coerce the data frame into a matrix first and you might get unexpected results.

Array – we can generalize the rules for vectors to arrays

An array is like a matrix, but with arbitrary dimension. The first argument is still a vector of elements to fill the array, but the second argument is a vector of sizes in each dimension

```
ar = array(1:24, dim = c(2, 3, 4))  # A 3-D array
ar
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18

, , 4

     [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24
ar[, , 1]  # extracting out the 1st block
     [,1] [,2] [,3]
[1,]    1    3    5
```

```
[2,]    2    4    6
ar
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18

, , 4

     [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24
ar[2, , 3:4]  # extracting out the 2nd row of the 3rd and 4th block
     [,1] [,2]
[1,]   14   20
[2,]   16   22
[3,]   18   24
ar
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

     [,1] [,2] [,3]
[1,]   13   15   17
```

```
[2,]   14   16   18

, , 4

     [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24
ar[1:2, 2:3, 3:4]  # extracting out the first 2 rows and
, , 1

     [,1] [,2]
[1,]   15   17
[2,]   16   18

, , 2

     [,1] [,2]
[1,]   21   23
[2,]   22   24
# the 2nd and 3th columns of block 3 and 4
```

We will not use arrays much in this class.

List and data frame

In terms of *element data types* data frames and lists are in the same group. Recall that a data frame is just a special case of a list: all the elements on a data frame need to be of the same length while elements of a list can be of various length.

In terms of *structures* data frames and matrices are in the same group and vectors and lists are in another group.

Compare a data frame with a matrix: Both have a similar structure; however, the columns (i.e., the elements) of a data frame can have different data types whereas the columns of a matrix have to be of the same data type.

Similarly, vectors and lists have a similar structure; however, the elements of a list can have different data types whereas that of a vector have to be of the same data type.

```
d = data.frame(let = c("a", "p"), val = 1:2, t = c(7.6, 5.1))

d
  let val   t
1   a   1 7.6
2   p   2 5.1
names(d)  # names of the elements of a data frame
[1] "let" "val" "t"
colnames(d)  # Remember that this also works for a matrix
[1] "let" "val" "t"
```

Indexing, extraction and concatenation

Data frame (this part is a review)

You can index a data frame with all the ways that work for matrices. However, different extraction method might give you an output of different data type. In addition, you can extract a column by name using the $ symbol.

```
class(d$val)  # a vector
[1] "integer"
class(d[[2]])  # a vector
[1] "integer"
class(d[, 2])  # a vector
[1] "integer"
class(d[2])  # a data frame
[1] "data.frame"

# Note that it makes sense to do this
class(d[2:3])
[1] "data.frame"
# but not this class(d[[2:3]]) see details here
# (https://cran.r-project.org/doc/manuals/R-lang.html#Indexing)
```

To combine multiple data frames we use the `data.frame()` function

```
# create another data frame to combine with d
d2 = data.frame(let2 = c("k", "j"), val2 = 8:7)
d2
  let2 val2
1    k    8
2    j    7

# combine two data frames

data.frame(d, d2)
  let val   t let2 val2
1   a   1 7.6    k    8
2   p   2 5.1    j    7
class(data.frame(d, d2))  # result is another data frame
[1] "data.frame"

cbind(d, d2)
  let val   t let2 val2
1   a   1 7.6    k    8
2   p   2 5.1    j    7
class(cbind(d, d2))
[1] "data.frame"
```

List

```
ingredients <- list(cheese = c("Cheddar", "Swiss", "Brie"), meat = c("Ham",
    "Turkey"))
ingredients
$cheese
[1] "Cheddar" "Swiss"   "Brie"

$meat
[1] "Ham"    "Turkey"

ingredients$meat  # Lists can be indexed by name, using $.
[1] "Ham"    "Turkey"
class(ingredients$meat)
[1] "character"
ingredients[2]
$meat
[1] "Ham"    "Turkey"
# can also be indexed like vectors, using [].  The result
# will be another list.
ingredients[[2]]
[1] "Ham"    "Turkey"
# can also extract individual elements of a list by using
# [[]]; recall the analogy of taking the R object out of a
# bag
```

To combine multiple lists we use the `c()` function

```
ingredients2 = list(veggies = c("lettuce", "tomato"))

c(ingredients, ingredients2)
$cheese
[1] "Cheddar" "Swiss"   "Brie"

$meat
[1] "Ham"    "Turkey"

$veggies
[1] "lettuce" "tomato"

# What happens if you use `list()` instead?
list(ingredients, ingredients2)
[[1]]
[[1]]$cheese
[1] "Cheddar" "Swiss"   "Brie"

[[1]]$meat
[1] "Ham"    "Turkey"
```

```
[[2]]
[[2]]$veggies
[1] "lettuce" "tomato"
```

Let's try out some exercises

```
head(airquality, n = 8)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
7    23     299  8.6   65     5   7
8    19      99 13.8   59     5   8
dim(airquality)
[1] 153   6
airquality[2, 3]
[1] 8
head(airquality$Wind)
[1]  7.4  8.0 12.6 11.5 14.3 14.9
head(airquality[, 3])  # same column of the dataset
[1]  7.4  8.0 12.6 11.5 14.3 14.9
# how many pairs of elements on these two vectors are
# unequal?
sum(airquality$Wind != airquality[, 3])
[1] 0
```

Summary: The types of R object and how to index them:

```
## Below are some examples; do not run them since you will
## need to create the objects for each of the types first

## Vectors: [index]
x[1:4]
x[-3]
x[x > 3]

## Matrices: [rowindex, colindex]
m[1, 2]
m[1:2, ]
m[, m[1, ] >= 2]
m["r1", ]
# here 'r1' is the row name for the first row of m

## Arrays: [index1, index2, ..., indexK]
a[1, 3, ]
```

```
a[a >= 6]

## Data frames: [rowindex, colindex], $name
cars$price
cars[, 3:4]
cars[cars$passengers == 5, ]

## Lists: $name, [index], [[index]]
ingredients$meat
ingredients[1:2]
ingredients[[1]]

## Note: both $ and [[]] can index only one element.
```

Some built-in functions in R

Before talking about `tapply()` and `table()` we should get ourselves familiar with the data type `factor`. Factors in R are categorical variables.

For example, I have 8 patients, some of them are in a control group whereas other ones are in a treatment group. I can create a vector `grp` that gives the group labels of these patients.

```
grp = factor(rep(c("Control", "Treatment"), each = 4))
```

```
grp
[1] Control   Control   Control   Control   Treatment Treatment
[7] Treatment Treatment
Levels: Control Treatment
```

Now I know that the first 4 patients are in the control group and the last 4 are in the treatment group. Note that a factor vector is interpreted differently by R than a character vector; in our case `Control` and `Treatment` in `grp` are categories, not characters.

---

tapply()

The `tapply()` function allows us to apply a function to different parts of a vector, where the parts are indexed by a factor or a list of factors.

Single factor:

```
effect = rnorm(8)  # Generate some fake data
effect
[1] -0.1528423 -0.3484571  0.1831076  0.2456646 -0.3474307
[6]  0.3806796 -1.2525991  0.5552255

`?`(tapply)
```

```
tapply(X = effect, INDEX = grp, FUN = mean)
    Control   Treatment
-0.01813179 -0.16603120
# `X` is the vector that you want to apply `FUN` to; `INDEX`
# is the vector used to subset `X`
```

Multiple factors:

```
sex = factor(rep(c("Female", "Male"), times = 4))
sex
[1] Female Male   Female Male   Female Male   Female Male
Levels: Female Male

tapply(effect, INDEX = list(grp, sex), FUN = mean)
               Female        Male
Control    0.01513267 -0.05139625
Treatment -0.80001492  0.46795253

# Here tapply evaluates mean(effect) on every group with
# unique combination of (grp, sex).
```

When using the `tapply()` function, you can include additional arguments for the function that you supply to `FUN` in the following way.

Let's say that there are some NAs in the vector effect.

```
effect.with.NAs = effect
effect.with.NAs[c(2, 5)] = NA
effect.with.NAs
[1] -0.1528423         NA  0.1831076  0.2456646         NA
[6]  0.3806796 -1.2525991  0.5552255
```

This will just give us NA outputs because `R` does not know how we would want to deal with the `NA`s.

```
tapply(effect.with.NAs, INDEX = grp, FUN = mean)
  Control Treatment
       NA        NA
```

Recall that `na.rm` is an input argument for `mean()`. We can set `na.rm = T` for `mean()` this way.

```
tapply(effect.with.NAs, INDEX = grp, FUN = mean, na.rm = T)
    Control   Treatment
 0.09197664 -0.10556468
```

The **table()** function uses the cross-classifying factors to build a contingency table of the counts at each combination of factor levels.

```
grp
[1] Control   Control   Control   Control   Treatment Treatment
[7] Treatment Treatment
Levels: Control Treatment

table(grp)
grp
  Control Treatment 
        4         4 
# counts how many elements you have in each group
```

`table()` is just a special case of `tapply()`.

```
tapply(rep(1, times = length(grp)), INDEX = grp, FUN = sum)
  Control Treatment 
        4         4 
# same result as above
```

A word of caution: the outputs of `table()` and `tapply()` are often arrays. However, a lot of functions often require their input arguments to be vectors. To convert an array into a vector we can use the `as.vector()` function. However, `as.vector()` strips the element names of its input by default. Therefore, we have to assign the names back to the vector. Here is an example

```
grp.table = table(grp)
class(grp.table)
[1] "table"
`?`(table)
# an object of class 'table' is an array of integer values
grp.table
grp
  Control Treatment 
        4         4 
as.vector(grp.table)  # element names were
[1] 4 4
# stripped

v.grp.table = as.vector(grp.table)
# assign element names back to the vector
names(v.grp.table) = names(grp.table)
```

`merge()`

```
# create data for the example
authors <- data.frame(surname = I(c("Tukey", "Venables", "Tierney",
    "Ripley", "McNeil")), nationality = c("US", "Australia",
    "US", "UK", "Australia"), deceased = c("yes", rep("no", 4)))
```

```r
books <- data.frame(name = I(c("Tukey", "Venables", "Tierney",
    "Ripley", "Ripley", "McNeil", "R Core")), title = c("Exploratory Data Analysis",
    "Modern Applied Statistics ...", "LISP-STAT", "Spatial Statistics",
    "Stochastic Simulation", "Interactive Data Analysis", "An Introduction to R"),
    other.author = c(NA, "Ripley", NA, NA, NA, NA, "Venables & Smith"))
```

```
authors
   surname nationality deceased
1    Tukey          US      yes
2 Venables   Australia       no
3  Tierney          US       no
4   Ripley          UK       no
5   McNeil   Australia       no
books
      name                         title     other.author
1    Tukey     Exploratory Data Analysis             <NA>
2 Venables Modern Applied Statistics ...           Ripley
3  Tierney                     LISP-STAT             <NA>
4   Ripley            Spatial Statistics             <NA>
5   Ripley         Stochastic Simulation             <NA>
6   McNeil     Interactive Data Analysis             <NA>
7   R Core          An Introduction to R Venables & Smith
```

```r
# Which data frame goes first?  If the variable for joining
# has different names the variable name in the data frame for
# the first input will be used.  Tje resulting data frame
# will have the overlap of `authors$surname` & `books$name`
authors.first = merge(x = authors, y = books, by.x = "surname",
    by.y = "name")

authors.first
   surname nationality deceased                         title
1   McNeil   Australia       no     Interactive Data Analysis
2   Ripley          UK       no            Spatial Statistics
3   Ripley          UK       no         Stochastic Simulation
4  Tierney          US       no                     LISP-STAT
5    Tukey          US      yes     Exploratory Data Analysis
6 Venables   Australia       no Modern Applied Statistics ...
  other.author
1         <NA>
2         <NA>
3         <NA>
4         <NA>
5         <NA>
```

```
6       Ripley
```

```
books.first = merge(x = books, y = authors, by.y = "surname",
    by.x = "name")

books.first
      name                         title other.author
1   McNeil     Interactive Data Analysis         <NA>
2   Ripley            Spatial Statistics         <NA>
3   Ripley         Stochastic Simulation         <NA>
4  Tierney                     LISP-STAT         <NA>
5    Tukey     Exploratory Data Analysis         <NA>
6 Venables Modern Applied Statistics ...       Ripley
  nationality deceased
1   Australia       no
2          UK       no
3          UK       no
4          US       no
5          US      yes
6   Australia       no
dim(authors)
[1] 5 3
dim(books)
[1] 7 3
```

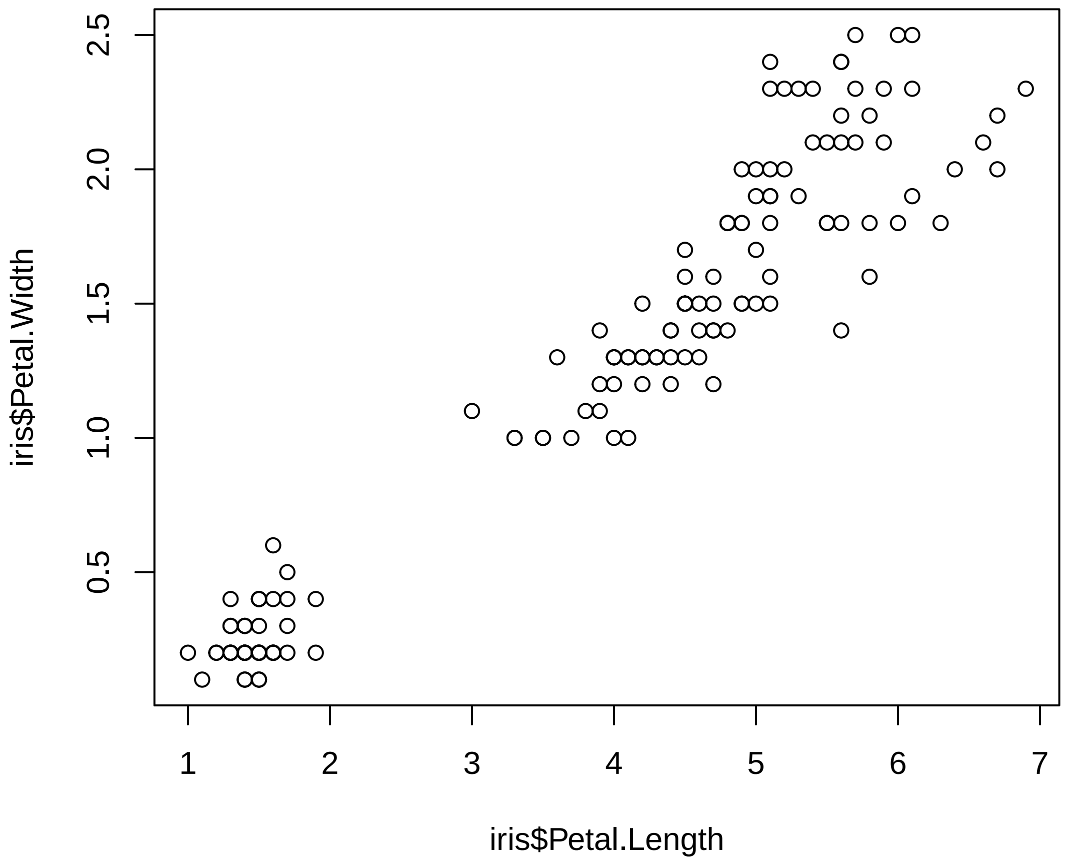
```
# check to see whether the names are in the same order note
# that the names are character vectors

sum(authors.first$surname != books.first$name)
[1] 0


# note that R Core was dropped since it is not in authors To
# include R Core, we can do a union-join
merge(x = authors, y = books, by.x = "surname", by.y = "name",
    all = TRUE)
   surname nationality deceased                         title
1   McNeil   Australia       no     Interactive Data Analysis
2   R Core        <NA>     <NA>          An Introduction to R
3   Ripley          UK       no            Spatial Statistics
4   Ripley          UK       no         Stochastic Simulation
5  Tierney          US       no                     LISP-STAT
6    Tukey          US      yes     Exploratory Data Analysis
7 Venables   Australia       no Modern Applied Statistics ...
      other.author
```

```
1              <NA>
2 Venables & Smith
3              <NA>
4              <NA>
5              <NA>
6              <NA>
7            Ripley
# or do this: a right-join
merge(x = authors, y = books, by.x = "surname", by.y = "name",
    all.y = TRUE)
   surname nationality deceased                         title
1   McNeil   Australia       no     Interactive Data Analysis
2   R Core        <NA>     <NA>          An Introduction to R
3   Ripley          UK       no            Spatial Statistics
4   Ripley          UK       no         Stochastic Simulation
5  Tierney          US       no                     LISP-STAT
6    Tukey          US      yes     Exploratory Data Analysis
7 Venables   Australia       no Modern Applied Statistics ...
      other.author
1              <NA>
2 Venables & Smith
3              <NA>
4              <NA>
5              <NA>
6              <NA>
7            Ripley
```

`with()`

`with()` is a very convenient function to use when you need to refer to a dataset multiple time in a line of code. `with()` allows you to evaluate an R expression in an environment constructed from data. E.g., I would like to plot `Petal.Length` v.s. `Petal.Width` for the data in the `iris` dataset. From what we have learned so far we could use the code

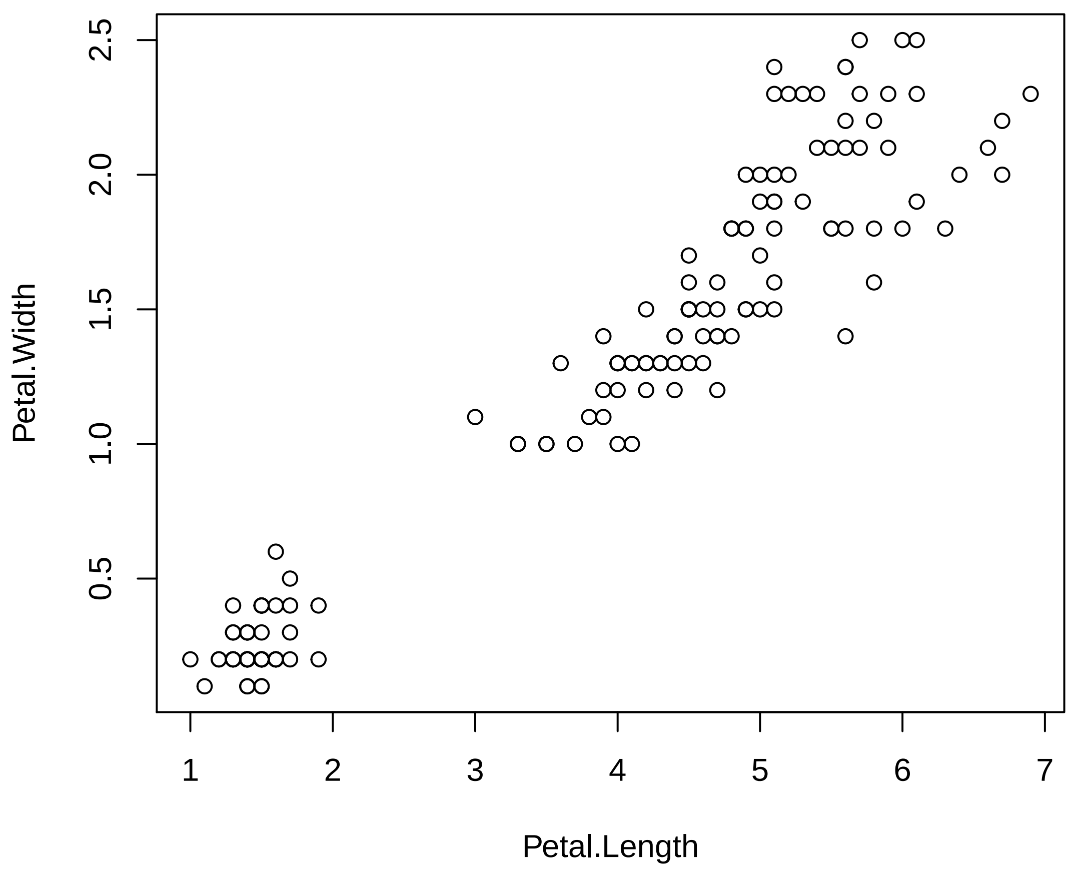
```
plot(x = iris$Petal.Length, y = iris$Petal.Width)
```



```
# we will skip labels for this example
```

The `with()` allows you to skip typing the name of the dataset for each of the `x-` and `y-` arguments:

```
with(iris, plot(x = Petal.Length, y = Petal.Width))
```



`mutate()`

`mutate()` in the `dplyr` package adds new variables to the input of the function and preserves existing variables; `mutate()` is a convenient function to use when you want to add new columns to an existing data frame.

```
# create variables to make a data frame
a = c("category I", "category II")
a
[1] "category I"  "category II"
m = matrix(1:6, ncol = 3)
m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
library("dplyr")
d = data.frame(a, m)
d
```

```
            a X1 X2 X3
1  category I  1  3  5
2 category II  2  4  6
# adding two new columns to d
new.d = mutate(d, X1minusX2 = X1 - X2, X2overX3 = X2/X3)
new.d
            a X1 X2 X3 X1minusX2  X2overX3
1  category I  1  3  5        -2 0.6000000
2 category II  2  4  6        -2 0.6666667
```

```
class(new.d)  #mutate preserve the object class
[1] "data.frame"
str(new.d)
'data.frame':   2 obs. of  6 variables:
 $ a        : Factor w/ 2 levels "category I","category II": 1 2
 $ X1       : int  1 2
 $ X2       : int  3 4
 $ X3       : int  5 6
 $ X1minusX2: int  -2 -2
 $ X2overX3 : num  0.6 0.667
```

This is the same as

```
new.d2 = data.frame(d, X1minusX2 = d$X1 - d$X2, X2overX3 = d$X2/d$X3)
new.d2
            a X1 X2 X3 X1minusX2  X2overX3
1  category I  1  3  5        -2 0.6000000
2 category II  2  4  6        -2 0.6666667
```

However, by default `data.frame()` will automatically convert all the character vector columns to factor vectors; thus, oftentimes it is better to use `mutate()` than `data.frame()`.

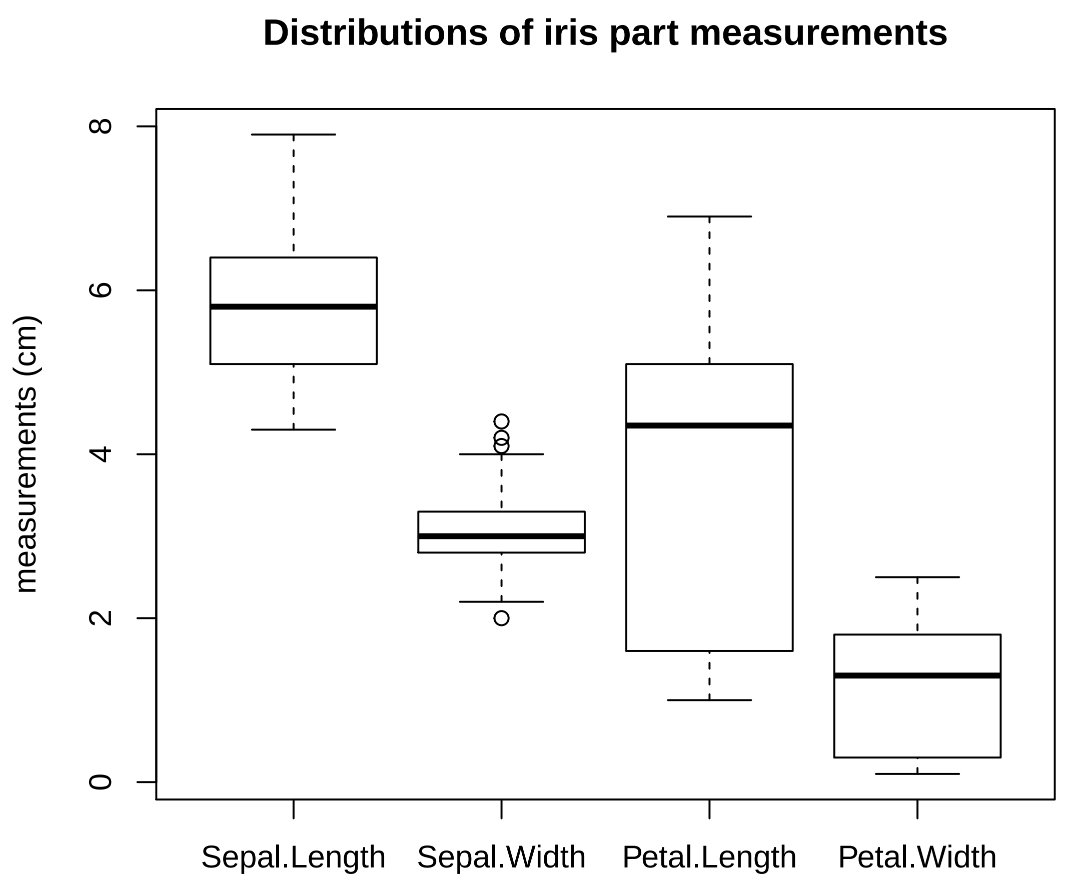
using `boxplot()` on a matrix

The function `boxplot()` can be used on a matrix to make multiple boxplots, one for each column of a matrix. For example, here we have four columns of a matrix that list the sepal length and width and petal length and width, respectively, for 150 iris.

```
flower.part.measurements = as.matrix(iris[, 1:4])
head(flower.part.measurements)
     Sepal.Length Sepal.Width Petal.Length Petal.Width
[1,]          5.1         3.5          1.4         0.2
[2,]          4.9         3.0          1.4         0.2
[3,]          4.7         3.2          1.3         0.2
[4,]          4.6         3.1          1.5         0.2
[5,]          5.0         3.6          1.4         0.2
[6,]          5.4         3.9          1.7         0.4
```

We can make four boxplots, one for each column of the matrix.

```
boxplot(flower.part.measurements, ylab = "measurements (cm)",
    main = "Distributions of iris part measurements")
```



Anatomy of a function

The syntax for writing a function is

function(input_argument_list){body}

- The keyword `function` just tells R that you want to create a function.
- The arguments to a function are its inputs, which may have default values.
- Next we have the body of the function, which typically consists of expressions surrounded by curly brackets. Think of these as performing some operations on the input values given by the arguments.

Typically we assign the **function** to a particular **name**. This should describe what the function does.

MyFunction = function(input_argument_list){body}

A function without a name is called an *orphan* function. These can be very powerfully used with the apply mechanism. Stay tuned. . . .

---

the **body of the function**, which typically consists of expressions surrounded by curly brackets. Think of these as performing some operations on the input values given by the arguments.

{ expression 1
expression 2
return(value)
}

The **return** expression controls the values that you want to output. If the function returns more than one thing, this is done using a named list, for example return(list(total = sum(x), avg = mean(x))).

In the **absence of a return expression**, a function will return the last evaluated expression. This is particularly common if the function is short.

For example, I could write the simple function:

MyMean = function(x) sum(x)/length(x)

Here I don't even need brackets {}, since there is only one expression.

Below is another example of a function. This function extracts out the most popular name for a given year and a given gender.

```
library(babynames)

popular.name = function(year, gender) {
    # year needs to be numeric `gender` need to be a character
    # vector of length one

    baby.year = babynames[babynames$year == year & babynames$sex ==
        gender, ]
    return(baby.year$name[baby.year$n == max(baby.year$n)])
}

popular.name(year = 2010, gender = "F")
[1] "Isabella"
```

A return expression anywhere in the function will cause the function to return control to the user immediately, without evaluating the rest of the function.

You will make a function to carry out these three steps for any data frame `d` in this week's precept.

```
dim(d)
class(d)
summary(d)
```

Environments and variable scope

R has a special mechanism for allowing you to use the same name in different places in your code and have it refer to different R objects. For example, you want to be able to create new variables in your functions and not worry about if there are variables with the same name already in the environment outside of the function.

Global v.s. function environment

When you call a function, R creates a new environment containing just the variables defined by the arguments of that function. If R cannot find a variable in the function environment, it will look for the variable again in the parent environment which will be the main workspace in the example below and is called the Global Environment–if you knit this the global environment would be the environment of your Rmarkdown file.

```
adding2 = function(x, y) {
    x + y
    print(ls())  # list variables in the function environment
    print(aa)
}

# adding2(x=-2, y=-1) this line can only be run in the
# console.  the error will prevent you from knitting if you
# uncomment it for the Rmarkdown file.
```

```
aa = 24  # define a in the global environment


# R still finds its way to `aa` even though `aa` is not in
# the function environment
adding2(x = -2, y = -1)
[1] "x" "y"
[1] 24
```

```
adding2 = function(x, y) {
    x + y
    print(ls())
```

```r
    # list variables in the function environment
    aa = 5
    print(aa)
}

# `aa` is in the function environment
adding2(x = -2, y = -1)
[1] "x" "y"
[1] 5
```