Data Analysis with R

R is a programming language and free software environment for statistical computing and graphics. R provides an open source route to easily apply a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, $\cdots$) and graphical techniques, and is highly extensible. R has one of the richest ecosystems to perform data analysis. There are around 12000 packages available in CRAN (open-source repository). The rich variety of library makes R the first choice for statistical analysis, especially for specialized analytical work.

In this workshop participants will take their first steps with R. The objective of this workshop will focus on creating new variables to conduct basic calculations, getting data to R, conducting simple statistics and visualization with R, and then introducing linear regression and decision tree models with applications in R.

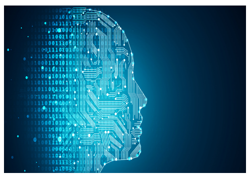
Workshop Facilitator: Jie (Jane) Jian
Date: April 27, Saturday
Time: 10:00 am - 12:00 pm
Venue: West Mall, Simon Fraser University

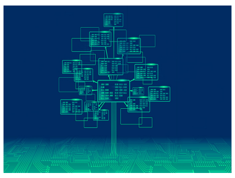
Main Topics

- Basic commands
- Loading data into R
- Data visualization with R
- Linear regression model
- Decision tree model

R can be downloaded from `https://www.r-project.org/`.

Advertising data can be downloaded from `http://www-bcf.usc.edu/~gareth/ISL/data.html`.
Iris data can be downloaded from `http://archive.ics.uci.edu/ml/datasets/iris`.



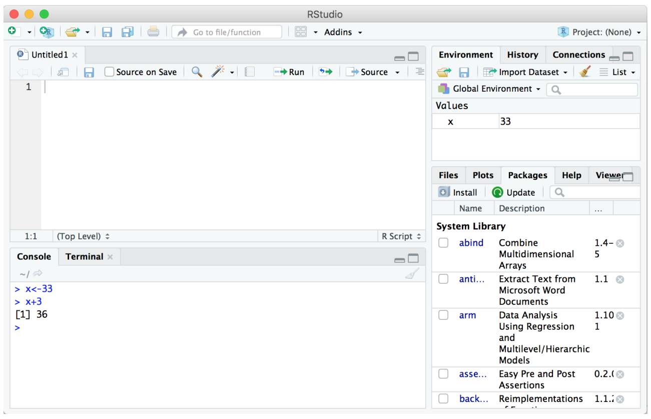


British Columbia Undergraduate Mathematics and Statistics Conference
Data Analysis with R
Jie (Jane) Jian
27th April, 2019

1 Basic Commands

Editor and Console

On the user interface, the bottom left panel is the console (or command line) where you directly execute commands and R writes results to the console. The > symbol is R's prompt for you to type something in the console. For example, type in x < − 33 hit return and then type x+3 and hit return again and watch the console:



```
> x<-33
> x+3
[1] 36
>
```

Values

x	33

System Library

	Name	Description	
☐	abind	Combine Multidimensional Arrays	1.4-5
☐	anti…	Extract Text from Microsoft Word Documents	1.1
☐	arm	Data Analysis Using Regression and Multilevel/Hierarchic Models	1.10-1
☐	asse…	Easy Pre and Post Assertions	0.2.(
☐	back…	Reimplementations	1.1.:

```
> x <- 33
> x+3
[1] 36
```

A basic concept in (statistical) programming is called a variable. A variable allows you to store a value (e.g. 33) or an object (e.g. a function description) in R. You can then later use this variable's name to easily access the value or the object that is stored within this variable. In the first line of the above block, we assign a value 33 to a variable called "x" with the assignment operator < −. In R, we can also use << −, = as the assignment operator. The operator < − can be used anywhere, whereas = sometimes does not serve as assignment operator. In the second line, we calculate x+3. Executing R commands straight in the console is a good way to experiment with R code, as your submission is not checked for correctness.

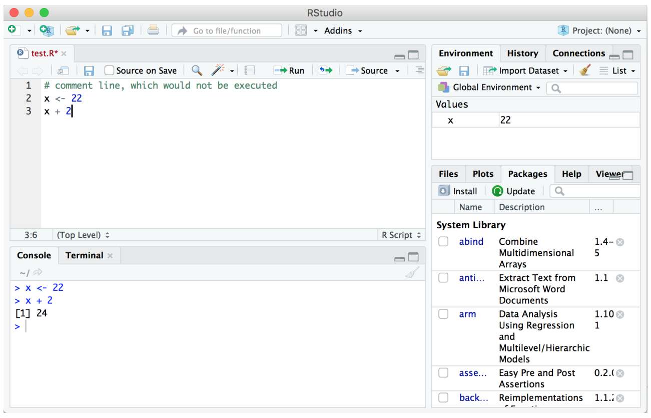
On the user interface, the top left panel is a text-box where you can write a script. When you hit the 'Submit Answer' button, every line of code is interpreted and executed by R and you get a message whether or not your code was correct. The output of your R code is shown in the console in the lower corner:

British Columbia Undergraduate Mathematics and Statistics Conference

Data Analysis with R

Jie (Jane) Jian

27th April, 2019



```
# comment line, which would not be executed
x <- 22
x + 2
```

```
> x <- 22
> x + 2
[1] 24
>
```

Values

x	22

	Name	Description	...
System Library			
☐	abind	Combine Multidimensional Arrays	1.4-5
☐	anti...	Extract Text from Microsoft Word Documents	1.1
☐	arm	Data Analysis Using Regression and Multilevel/Hierarchic Models	1.10 1
☐	asse...	Easy Pre and Post Assertions	0.2.(
☐	back...	Reimplementations	1.1.:

R makes use of the # sign to add comments, so that you and others can understand what the R code is about. Comments are not run as R code, so they will not influence your result.

Create a Vector and a Matrix

R uses functions to perform operations. A function can have some number of inputs. For example, to create a vector of numbers, we use the function c(). Any number inside the parentheses are joined together. The following commands in the console instruct R to join together the numbers 1, 2, 3, and 4, and to save them as a vector named x.

```
> y <- c(1,2,3,4)
> y
[1] 1 2 3 4
```

You can assign another data type to the variable y. To compound things further you can even change the data type to strings overwriting y.

```
> y <- c("Toronto","Montreal","Vancouver")
> y
[1] "Toronto"   "Montreal"   "Vancouver"
```

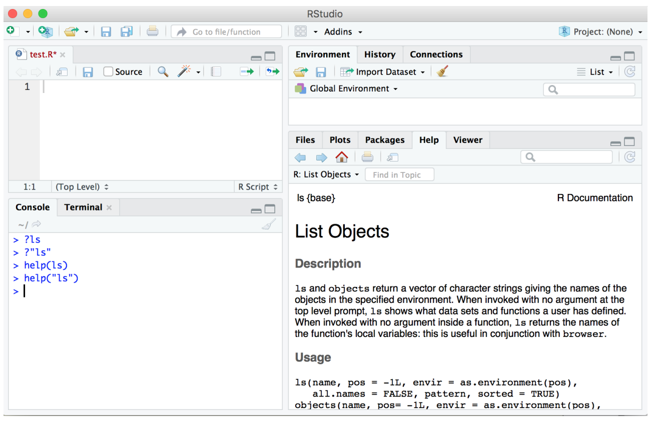
To access individual elements of a vector, in this case the third element, we execute:

```
> y[3]
[1] "Vancouver"
```

We can check the length of a vector using the length() function.

```
> length(y)
[1] 3
```

The ls() function allows us to look at a list of all the objects, such as data and functions, that we saved so far. The rm() function can be used to delete any that we don't want. Before we use the ls() function, we can learn more about it by R help. The help() function and ? help operator in R provide access to the documentation pages for R functions, data sets, and other objects, both for packages in the standard R distribution and for contributed packages. To access documentation for the ls() function, for example, enter the command help(ls) or help("ls"), or ?lm or ?"lm" (i.e., the quotes are optional).



```
> ?ls
> ?"ls"
> help(ls)
> help("ls")
>
```

ls {base} R Documentation

List Objects

Description

`ls` and `objects` return a vector of character strings giving the names of the objects in the specified environment. When invoked with no argument at the top level prompt, `ls` shows what data sets and functions a user has defined. When invoked with no argument inside a function, `ls` returns the names of the function's local variables: this is useful in conjunction with `browser`.

Usage

```
ls(name, pos = -1L, envir = as.environment(pos),
   all.names = FALSE, pattern, sorted = TRUE)
objects(name, pos= -1L, envir = as.environment(pos),
```

```
> ls()
[1] "x" "y"
> rm(x)
> ls()
[1] "y"
```

The matrix() function takes a number of inputs and creates a matrix from the given set of values. We can have first three inputs as the data (the entries in the matrix), the number of rows, and the number of columns. As we can see from the example matrix(c(1,2,3,4,5,6,7,8,9),3,3) or matrix(1:9,3,3), number 1 to number 9 has been shaped in a 3×3 matrix. But if we want to have number 1 to number 9 has been shaped in a 3×4 matrix, we would have a warning shown that in matrix(1:9, 3, 4) : data length [9] is not a sub-multiple or multiple of the number of columns [4]. By default, R creates matrices by successively filling in columns. If the option byrow=TRUE is set, then the matrix will be generated in order of the rows.

```
> matX <- matrix(c(1,2,3,4,5,6,7,8,9),3,3)
> matX
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> matY <- matrix(1:9,3,3)
> matY
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> matZ <- matrix(c(1,2,3,4,5,6,7,8,9),3,3,byrow = TRUE)
> matZ
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

The sqrt() function returns the square root of each element of a vector or matrix. The command x^2 raises each element of matX to the power 2.

```
> sqrt(matX)
         [,1]     [,2]     [,3]
```

```
[1,]  1.000000  2.000000  2.645751
[2,]  1.414214  2.236068  2.828427
[3,]  1.732051  2.449490  3.000000
> matX^2
      [,1]  [,2]  [,3]
[1,]     1    16    49
[2,]     4    25    64
[3,]     9    36    81
```

Sometimes, we want to examine part of a set of data. For example, we want to refer to the element corresponding to the second row and the third column of matrix matX, then typing the following:

```
> matX
      [,1]  [,2]  [,3]
[1,]     1     4     7
[2,]     2     5     8
[3,]     3     6     9
> matX[2,3]
[1]  8
```

The first number after the open-brackets symbol refers to the row index and the second number refers to the column. We can also select multiple rows and columns at a time:

```
> matX
      [,1]  [,2]  [,3]
[1,]     1     4     7
[2,]     2     5     8
[3,]     3     6     9
> matX[c(2:3),c(1:3)]
      [,1]  [,2]  [,3]
[1,]     2     5     8
[2,]     3     6     9
> matX[2:3,1:3]
      [,1]  [,2]  [,3]
[1,]     2     5     8
[2,]     3     6     9
> matX[c(2,3),c(1,2,3)]
      [,1]  [,2]  [,3]
[1,]     2     5     8
[2,]     3     6     9
> matX[1,]
[1]  1  4  7
> matX[,2:3]
      [,1]  [,2]
[1,]     4     7
[2,]     5     8
[3,]     6     9
```

Using the dim(A) function outputs the size of a given matrix by showing the number of row and number of column.

```
> matX[,2:3]
      [,1]  [,2]
[1,]     4     7
[2,]     5     8
```

```
[3,]    6    9
> dim(matX[,2:3])
[1] 3 2
```

To generate a vector in which each element is a standard normal random variable, we can use the rnorm() with the first argument n the sample size. We will get different answer each time we call this function. By default, rnorm() creates standard normal random variables with a mean of 0 and a standard deviation of 1. To change the mean and standard deviation, one can use the arguments mean and sd.

Here we create two correlated sets of numbers, x and y, and use the cor() function to compute the correlation between them.

```
> x <- rnorm(50)
> y <- x+rnorm(50,mean=50,sd=0.1)
> cor(x,y)
[1] 0.9936963
```

If we want our code to reproduce the exact same set of random numbers, we can easily use the set.seed() function to do this. The argument in the set.seed() function can be taken as any arbitrary integer.

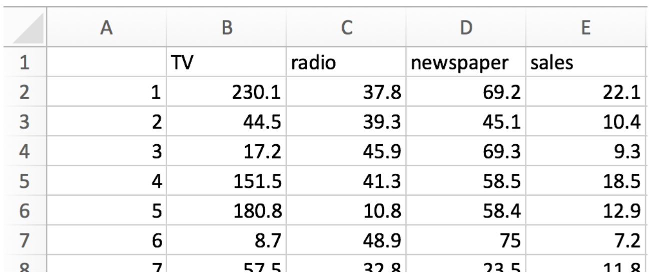
```
> set.seed(1234)
> rnorm(10)
 [1] -1.2070657  0.2774292  1.0844412 -2.3456977  0.4291247  0.5060559 -0.5747400
      -0.5466319 -0.5644520 -0.8900378
> set.seed(1234)
> rnorm(10)
 [1] -1.2070657  0.2774292  1.0844412 -2.3456977  0.4291247  0.5060559 -0.5747400
      -0.5466319 -0.5644520 -0.8900378
```

The mean() and var() functions can be used to compute the mean and variance of a vector of numbers. Using sqrt() to the output of var() will give the standard deviation. Alternatively, we can just use the sd() function to calculate the standard deviation of a vector.

```
> set.seed(222)
> x=rnorm(1000)
> mean(x)
[1] -0.01561228
> var(x)
[1] 0.9865457
> sqrt(var(x))
[1] 0.9932501
> sd(x)
[1] 0.9932501
```

2 Loading data into R

For most analysis, the first step is to import a data set into R. We begin with the Advertising data set consists of the sales of that product in 200 different markets, along with advertising budgets for the product in each of those markets for three media: TV, radio, and newspaper.



	A	B	C	D	E
1		TV	radio	newspaper	sales
2	1	230.1	37.8	69.2	22.1
3	2	44.5	39.3	45.1	10.4
4	3	17.2	45.9	69.3	9.3
5	4	151.5	41.3	58.5	18.5
6	5	180.8	10.8	58.4	12.9
7	6	8.7	48.9	75	7.2
8	7	57.5	32.8	23.5	11.8

We can easily download the Advertising.csv data from `http://www-bcf.usc.edu/~gareth/ISL/data.html`. Alternatively, the github of the workshop prepared the data for us, at `https://github.com/lfunderburk/BC-UMSC-Workshops/tree/master/Workshops/JaneJian`. Save the file to the same folder where we saved our R script. To further make sure that we are in the directory where the data was saved, we can select Session (under the menu bar) − > Set working directory − > To Source File Location or simply choose direction to the folder where we saved the dataset.

The following command read.table() will load the Advertising.csv file into R and store it as an object called ad_data. Using the option header=T (or header=TRUE) tells R that the first line of the file contains the variable names, and using the option sep="," tells R that values on each line of the file are separated by this character ,.

```
> ad_data <- read.table("Advertising.csv",header = TRUE, sep = ",")
> View(ad_data)
> dim(ad_data)
[1] 200    5
```

Double click the ad_data in the Environment window or input the command View(ad_data), then the dataset would be displayed to the screen. The dim() function shows that the size of the dataset is 200 by 5, which means that the data set includes 200 rows and 5 columns.

Since our data file is a csv file (comma separate value file), an easier way to load the file to R would be directly use the read.csv() function:

```
> ad_data <- read.csv("Advertising.csv")
```

We noticed that the first column of ad_data was the index of row, which can be removed. So we overwrite the ad_data to get rid of the first column. -1 in the second argument of the brackets means that we remove the first column of the data.

```
> ad_data <- ad_data[,-1]
```

After we loaded the data correctly, we can use names() function to check the variable names.

```
> names(ad_data)
[1] "TV"         "radio"      "newspaper" "sales"
```

3 Data visualization with R

The plot() function is a basic tool to plot data in R. For instance, we first generate two sets of standard normal random numbers, and then apply the plot() with some arguments to produce the scatterplot.

```
> x <- rnorm(100)
> y <- rnorm(100)
> plot(x,y)
> plot(x,y,xlab="x-axis",ylab="y-axis",main="Plot of X vs Y")
> plot(x,y,xlab="x-axis",ylab="y-axis",main="Plot of X vs Y",col="red")
```

Now, let's use the plot() function to produce the scatterplot of the advertising budgets of TV and the sales of that product. If we simply type the names of the variables into the function as plot(TV,sales), then we would get an error massage, because we didn't tell R to focus on the data set of ad_data for those variables.

```
> plot(TV, sales)
Error in plot(TV, sales) : object 'TV' not found
```

To tell R to look into the ad_data, there are three strategies.

Firstly, to refer a variable within a dataset, we can type the data set and the variable name joined with a $ symbol.

```
> plot(ad_data$TV, ad_data$sales)
```

Secondly, as we know that the budgets of TV is the first column and the sales is the fourth column, we can produce the scatterplot of the first and second column.

```
> plot(ad_data[,1], ad_data[,4])
```

Thirdly, we can use the attach() function in order to tell R to make focus on the given dataset. The attach() function is the perfect tool to pin a dataset when there is only one single dataset which will be frequently used. By this method, we can produce the pairwise scatterplots of budgets of TV and sales, budgets of radio and sales, budgets of newspaper and sales in a convenient way.

```
> attach(ad_data)
> plot(TV, sales)
> plot(radio, sales)
> plot(newspaper, sales)
```

The pair() function creates a scatterplot matrix, which is the scatterplot for every pair of variables for the given dataset. The following two commands give the same scatterplot.

```
> pairs(ad_data)
> pairs( ~ TV + radio + newspaper + sales)
```

The summary() function produces a numerical summary of each variable, including the minimum, maximum, the first quartile (Q_1), the third quartile (Q_3), median, and mean.

```
> summary(ad_data)
       TV             radio          newspaper          sales
 Min.   :  0.70   Min.   : 0.000   Min.   :  0.30   Min.   : 1.60
 1st Qu.: 74.38   1st Qu.: 9.975   1st Qu.: 12.75   1st Qu.:10.38
 Median :149.75   Median :22.900   Median : 25.75   Median :12.90
 Mean   :147.04   Mean   :23.264   Mean   : 30.55   Mean   :14.02
 3rd Qu.:218.82   3rd Qu.:36.525   3rd Qu.: 45.10   3rd Qu.:17.40
 Max.   :296.40   Max.   :49.600   Max.   :114.00   Max.   :27.00
```

4 Linear Regression Model

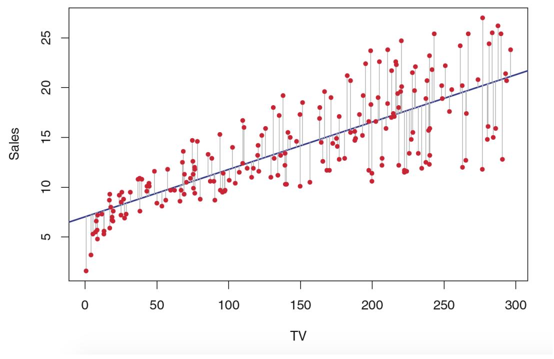
Introduction to Linear Regression Model

Based on the scatterplot of the budgets of TV and sales, we want to predict the sales on the budgets of TV. If X represent TV advertising and Y represents sales. Then we can regress sales onto TV by fitting the model

$$sales \approx \beta_0 + \beta_1 \times TV \tag{1}$$

In the equation, β_0 and β_1 are the two unknown constants that represent the intercept and slope terms in the linear model, which are called model coefficients or parameters. In real life, we do not know the model coefficients β_0 and β_1. We need to use the data to estimate the coefficients β_0 and β_1 such that the linear model $sales \approx \beta_0 + \beta_1 \times TV$ fits the data well. We want to find a straight line, which is as close as possible to all the data points.

There are multiple ways to measure the closeness. The most common method is sum of residual squares. To minimize the closeness between data points and the fitted straight line, we have to minimize the sum of residual square, and this approach is called least squares regression. As the following scatterplot shown, each grey line segment represents an error, and the fit makes a compromise by averaging their squares.



Linear Regression with R

In R, we can use the lm() function to fit a simple linear regression model. The basic syntax is lm(y ∼ x,data), where the fist argument y ∼ x represents that y is the response and x is the predictor, and the second argument data means that the dataset we used. In the advertising example, if we want to regress sales onto TV advertising:

```
> lm(ad_data$sales ~ ad_data$TV, ad_data)

Call:
lm(formula = ad_data$sales ~ ad_data$TV, data = ad_data)

Coefficients:
(Intercept)    ad_data$TV
    7.03259       0.04754

> lm.fit <- lm(ad_data$sales ~ ad_data$TV, ad_data)
> summary(lm.fit)

Call:
lm(formula = ad_data$sales ~ ad_data$TV, data = ad_data)

Residuals:
    Min      1Q  Median      3Q     Max
-8.3860 -1.9545 -0.1913  2.0671  7.2124

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 7.032594   0.457843   15.36   <2e-16 ***
ad_data$TV  0.047537   0.002691   17.67   <2e-16 ***
---
Signif. codes:  0 ?***? 0.001 ?**? 0.01 ?*? 0.05 ?.? 0.1 ? ? 1
```

```

Residual standard error: 3.259 on 198 degrees of freedom
Multiple R-squared:  0.6119,  Adjusted R-squared:  0.6099
F-statistic: 312.1 on 1 and 198 DF,  p-value: < 2.2e-16
```

If we simply type lm(ad_data$sales ∼ ad_data$TV, ad_data), we will get some basic information, such as the best-fit coefficients $\beta_0 = 7.03259$ and $\beta_1 = 0.04754$. Hence, the linear model to regress the sales on TV advertising is

$$sales = 7.03259 + 0.04754 \times TV + error. \tag{2}$$

For more detailed information, we assign the output of the linear regression model to a variable lm.fit. summary(lm.fit) gives us p-values and standard errors for the coefficients, as well as the R^2 statistic and F-statistic for the model.

We can use the names() function to find out what other pieces of information are stored in lm.fit.

```
> names(lm.fit)
 [1] "coefficients"  "residuals"     "effects"       "rank"          "fitted.values"
 [6] "assign"        "qr"            "df.residual"   "xlevels"       "call"
[11] "terms"         "model"
```

We can extract these quantities by their names, like lm.fit$coefficient. Alternatively, we can also use the function coef() to access the coefficient of the linear regression model.

```
> lm.fit$coefficients
(Intercept)          TV
 7.03259355  0.04753664
> coef(lm.fit)
(Intercept)          TV
 7.03259355  0.04753664
```

By using the confint() command, we can check the confidence interval for the coefficient estimates.

```
> confint(lm.fit)
                 2.5 %     97.5 %
(Intercept) 6.12971927 7.93546783
TV          0.04223072 0.05284256
```

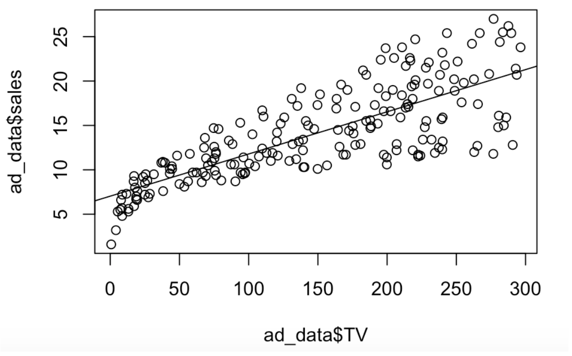
As we can see, the 95% confidence interval for β_0 is $[6.130, 7.935]$ and the 95% confidence interval for β_1 is $[0.042, 0.053]$. Therefore, we can conclude that in the absence of any advertising, sales will have a value between 6.130 and 7.940 units on average. Furthermore, for each $1,000 increase in television advertising, there will be an average increase in sales of between 42 and 53 units.

The predict() function can be used to produce confidence intervals and prediction intervals for the prediction of sales for a given value of TV. Suppose the budgets of TV advertising are 0.1, 1 and 10, to solve the predicted sales we have:

```
> predict(lm.fit,data.frame(TV=c(0,0.1,1,10)))
       1        2        3        4
7.032594 7.037347 7.080130 7.507960
```

We will now plot TV and sales along with the least square regression line using the abline() function. Since abline() has to be called on an existing plot. We can't call it when nothing has been plotted. So we will first plot the scatterplot of TV and sales.

```
> plot(TV, sales)
> abline(lm.fit)
```



Multiple Linear Regression

We want to know if there is relationship between advertising sales and budget, and how strong is the relationship. Let's first calculate the correlation matrix for TV, radio, newspaper and sales:

```
> cor(ad_data)
                  TV      radio  newspaper     sales
TV        1.00000000 0.05480866 0.05664787 0.7822244
radio     0.05480866 1.00000000 0.35410375 0.5762226
newspaper 0.05664787 0.35410375 1.00000000 0.2282990
sales     0.78222442 0.57622257 0.22829903 1.0000000
```

Notice that the correlation between sales and the three ways of media are positive. And we can consider that the advertising budgets of the three have an impact on the sales. Therefore, we can fit a multiple linear regression model using least squares, we again use the lm() function. The syntax lm(y$\sim x_1 + x_2 + x_3$) is used to fit a model with three predictors, TV, radio and newspaper. The summary function now outputs the regression coefficients for all the predictors.

```
> lm.fit3 <- lm(sales ~ TV + radio + newspaper)
> summary(lm.fit3)

Call:
lm(formula = sales ~ TV + radio + newspaper)

Residuals:
    Min      1Q  Median      3Q     Max
-8.8277 -0.8908  0.2418  1.1893  2.8292

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.938889   0.311908   9.422   <2e-16 ***
TV           0.045765   0.001395  32.809   <2e-16 ***
radio        0.188530   0.008611  21.893   <2e-16 ***
newspaper   -0.001037   0.005871  -0.177     0.86
---
```

```
Signif. codes:  0 ?***? 0.001 ?**? 0.01 ?*? 0.05 ?.? 0.1 ? ? 1

Residual standard error: 1.686 on 196 degrees of freedom
Multiple R-squared:  0.8972,  Adjusted R-squared:  0.8956
F-statistic: 570.3 on 3 and 196 DF,  p-value: < 2.2e-16
```

Hence, the multiple linear model to regress the sales on TV, radios and newpapers advertising is

$$sales = 2.938889 + 0.045765 \times TV + 0.188530 \times radio - 0.001037 \times newspaper + error. \tag{3}$$

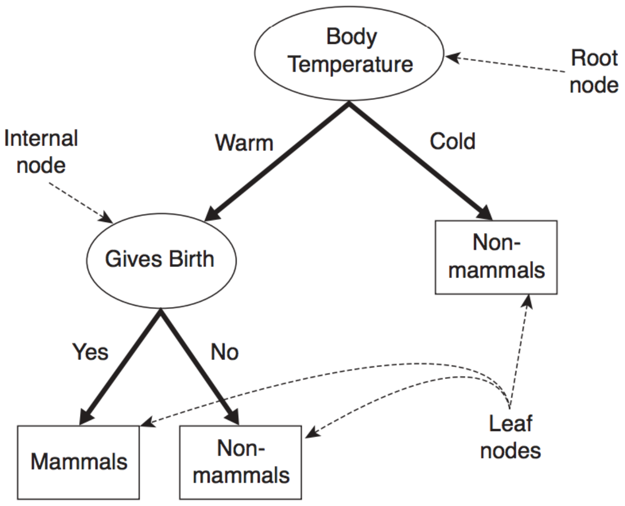
We interpret these results as follows: for a given amount of TV and newspaper advertising, spending an additional $1,000 on radio advertising leads to an increase in sales by approximately 189 units. Comparing these coefficient estimates to the simple regression coefficients, we notice that the multiple regression coefficient estimates for TV and radio are pretty similar to the simple linear regression coefficient estimates. However, while the newspaper regression coefficient estimate in multiple model is negative. This illustrates that the simple linear regression and the multiple linear regression can be very different.

5 Decision Tree

Introduction to Decision Tree

Imagine that we have to classify an animal into one of two categories: mammals or non-mammals.

Mammals or Non-mammals?



One effective approach may show as the above tree diagram that we ask ourselves a series of questions. First, whether the animal is cold- or warm-blooded. If it is cold-blooded, then it is definitely not a mammal. If it is warm blood, it is either a bird or a mammal. Following by the first question, the second would be: Do the females give birth to their young? If the answer is yes, then they are definitely mammals.

When we solve a classification problem by asking a series of questions, we are actually proceeding the decision tree model.

Decision Tree with R

For decision trees, load the rpart library, first installing it by install.packages("rpart") and then loading it with library(rpart).

The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2. Five attribute Information is:

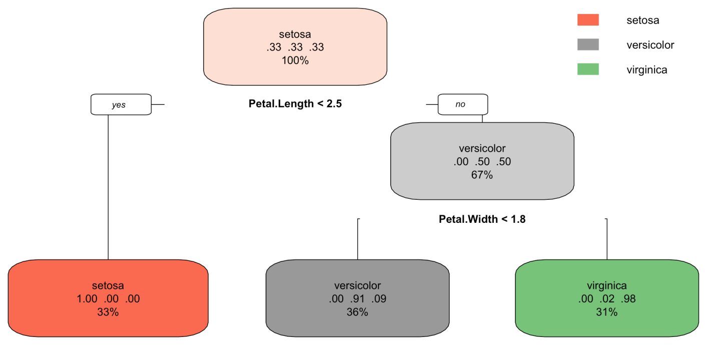
1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm
5. class:
   - Iris Setosa
   - Iris Versicolour
   - Iris Virginica

```
> library(rpart)
> data("iris")
> iris_data <- iris
> iris_data[1:5,]
  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
> names(iris_data)
 [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
> dectree <- rpart(Species ~ ., method = "class", data = iris_data)
> dectree
n= 150

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 150 100 setosa (0.33333333 0.33333333 0.33333333)
  2) Petal.Length< 2.45 50   0 setosa (1.00000000 0.00000000 0.00000000) *
  3) Petal.Length>=2.45 100  50 versicolor (0.00000000 0.50000000 0.50000000)
    6) Petal.Width< 1.75 54   5 versicolor (0.00000000 0.90740741 0.09259259) *
    7) Petal.Width>=1.75 46   1 virginica (0.00000000 0.02173913 0.97826087) *
> library(rpart.plot)
> rpart.plot(dectree)
```



References

[1] James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *The Elements of Statistical Learning.* Springer New York Inc., 2001.

[2] James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning.* New York: springer, 2013.

[3] Tan, Pang-Ning, Michael Steinbach, Vipin Kumar, and Anuj Karpatne. *Introduction to Data Mining, Global Edition.* Pearson Education Limited, 2019.