同濟大學
TONGJI UNIVERSITY

生物信息学系
DEPARTMENT OF BIOINFORMATICS

Section 14

R: Programming

张勇

yzhang@tongji.edu.cn

同济大学

2021年6月21日

R programming

- So far we can do:
  - Summarize and display data
  - Fit models to data & display results
  - Stochastic simulation
- With programming, we can:
  - Automate iterative tasks
  - Handle more complex data and modeling
  - Write custom functions
  - Modify existing R functions

for() **loop**

- **for( )** statement allows one to specify that a certain operation should be repeated a fixed number of times.

- Examples
  - compute the factorial of 20
  - stochastic simulations are very repetitive; we want to see patterns of behavior, not just a single instance.

for() **loop**

- Syntax

```
for (name in vector) {
    commands
}
```

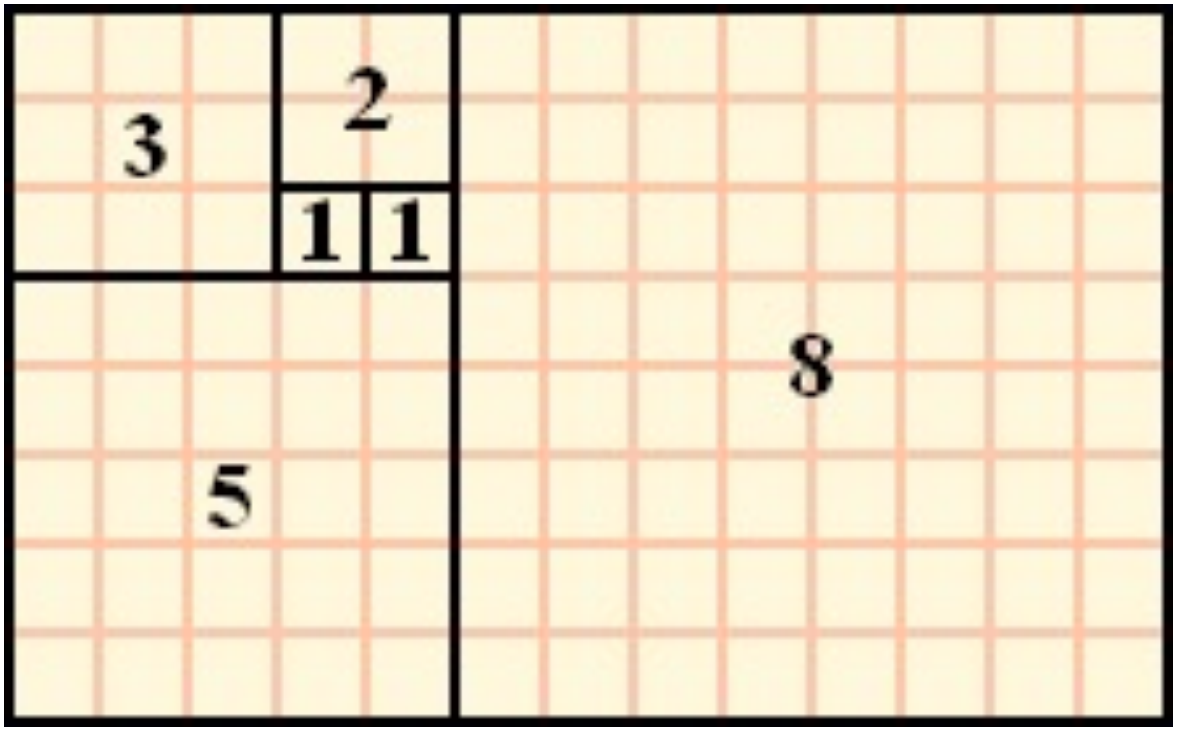
  - This sets a variable called **name** equal to each of the elements of **vector**, in sequence
  - For each value of **name**, the **commands** within the curly braces will be performed
    - For a single command, braces can be omitted

Example: Fibonacci sequence

- Considers the growth of a rabbit population, assuming that:
  - At month 1 there is one pair of newborn rabbits
  - After two months they reach puberty and can give birth to a new pair
  - All mature pairs give birth to a new pair monthly
  - Rabbits never die

- Total pairs: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55…
- F(n) = F(n-1) + F(n-2)

Fibonacci sequence



A tiling with squares whose sides are successive Fibonacci numbers in length

Fibonacci sequence

R code for Fibonacci sequence, using for()

```
Fibonacci <- numeric(12)

Fibonacci[1] <- Fibonacci[2] <- 1

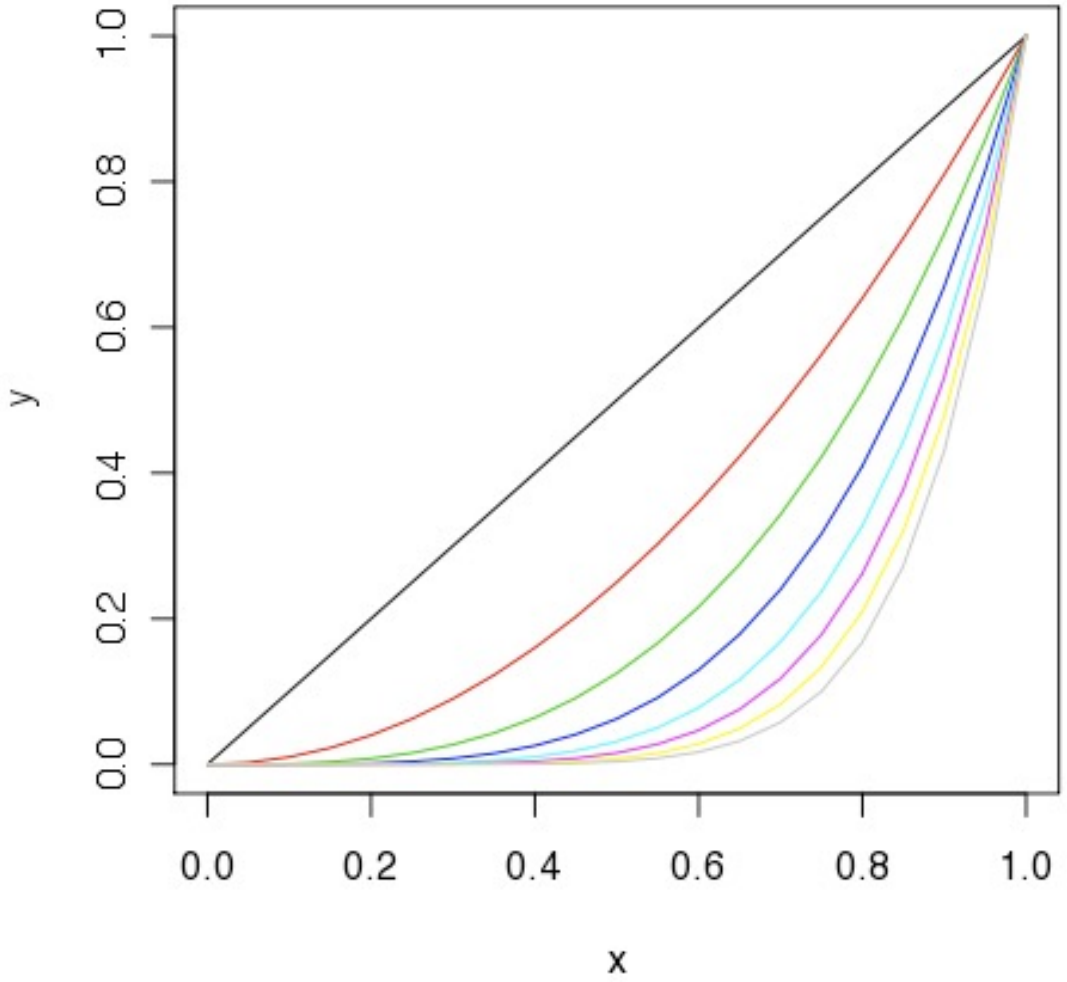
for (i in 3:12) {

   Fibonacci[i] <- Fibonacci[i - 2] + Fibonacci[i - 1]

}
```

for() **example**

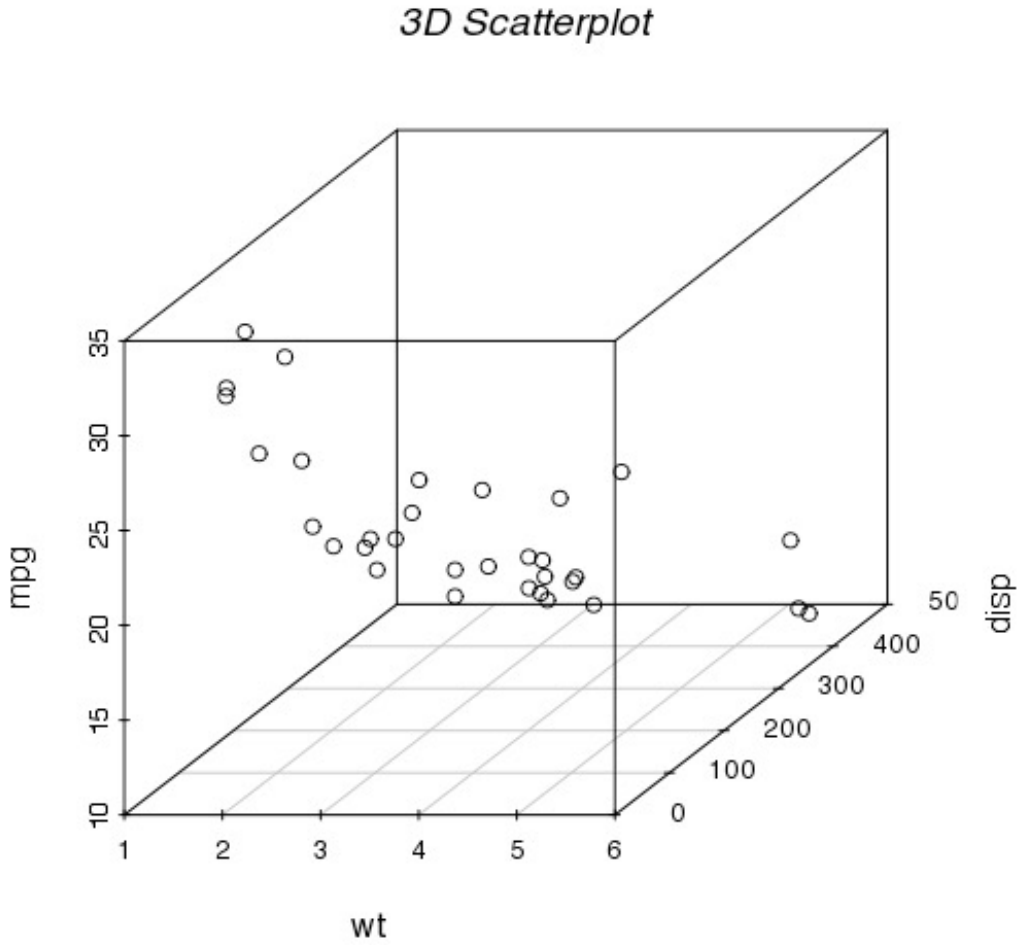
```
x <- seq(0, 1, .05)
plot(x, x, ylab="y", type="l")
for (i in 2:8)
    lines(x, x^i, col=i)
```



for() **example**

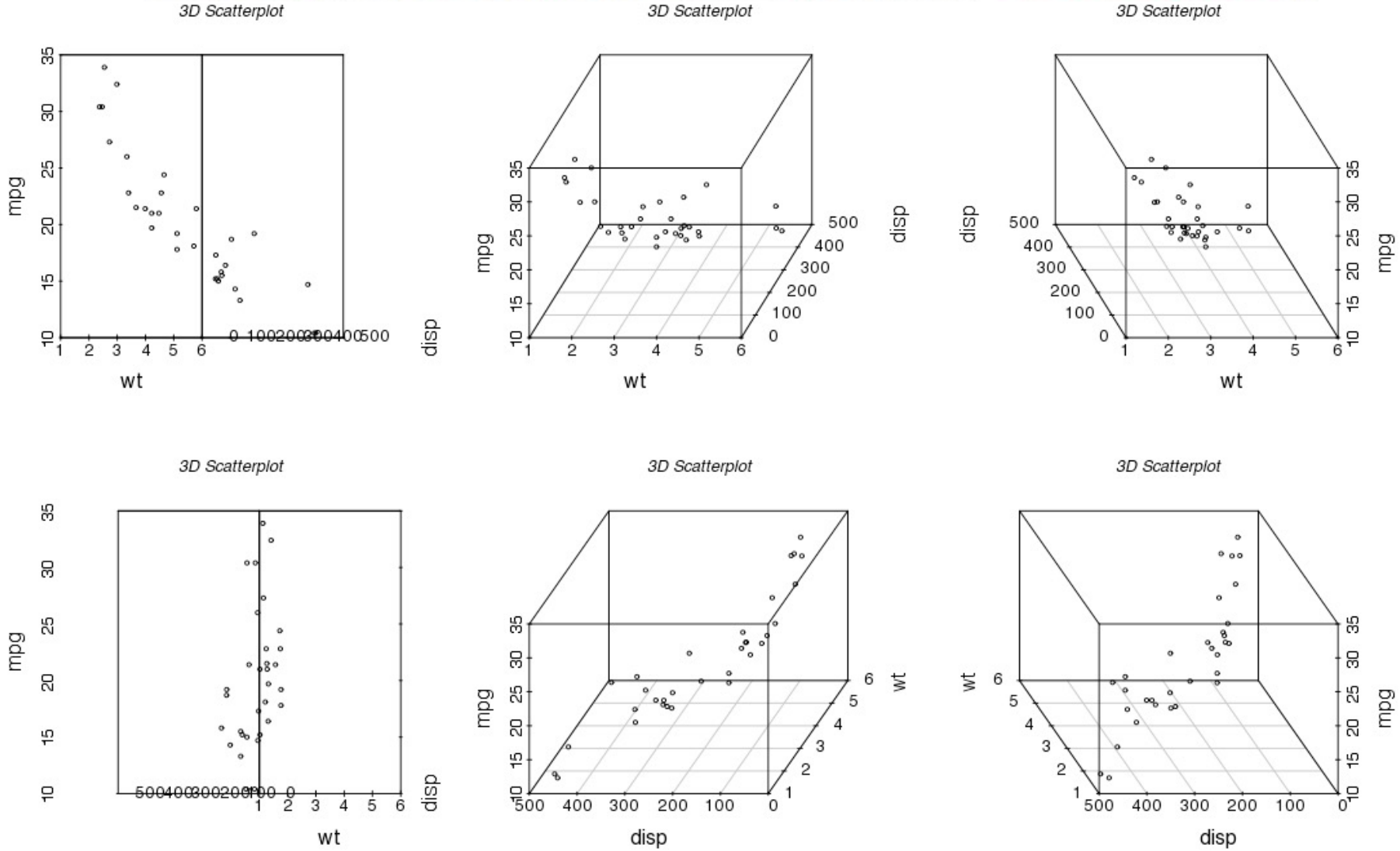
```
install.packages("scatterplot3d")
library(scatterplot3d)

attach(mtcars)
scatterplot3d(wt,disp,mpg, main="3D Scatterplot")
```



for() **example**

```
par(mfrow=c(2,3))
for (angle in seq(0, 300, 60))
    scatterplot3d(wt,disp,mpg, angle=angle, main="3D Scatterplot")
```



if() **statement**

- The **if( )** statement allows us to control which statements are executed, depending on the values of some input or variables.

- Examples
  - if it rains tomorrow, I will stay at home; if not, go to school.
  - `if (x > 2) y <- 2*x else y <- 3*x`

if() **statement**

- Syntax 1

```
if (condition) {
    commands when TRUE
}
```

- Syntax 2

```
if (condition) {
    commands when TRUE
} else {
    commands when FALSE
}
```

  - **condition** is logical expression of R, such as "x > 10"
  - Numerical values can be used as the value of **condition:** 0 is FALSE, non-zeros are TRUE.

if() **statement: code example**

- Find prime numbers up to n
  1. begin with a vector from 2 to n
  2. starting with 2, remove all multiples of 2 which are larger than 2
  3. move to the next number remaining in the vector, in this case, 3
  4. remove all multiples of 3 which are larger than 3
  5. proceed through all remaining entries of the vector in this way ……

if() **statement: code example**

```
n <- 50

sieve <- seq(2, n)   # sieve initially contains 2:n
primes <- c()        # empty vector

for (i in seq(2, n)) {

  if (any(sieve == i)) {    # sieve contains i
    primes <- c(primes, i)  # add i to prime list
    sieve <- c(sieve[(sieve %% i) != 0])  # multiples of i are removed
  }

}
```

One comment about if() format

```
x <- 2
if (x > 2) { print("x > 2") }
else {print("x <= 2") }
```

Wrong!

```
if (x > 2) { print("x > 2")
} else {print("x <= 2") }
```

Correct!

```
if (x > 2) {
    print("x > 2")
} else {
    print("x <= 2")
}
```

Better!

while() **loop**

- We want to repeat statements, but the pattern of repetition is not known in advance.
  - We need to do some calculations and keep going as long as a condition holds.

- Examples
  - while it rains, I will eat an orange; when it stops raining, finish eating.
  - ```
    while (x.total < 100)
      x.total <- x.total + runif(1)
    ```
    ```

while() **loop**

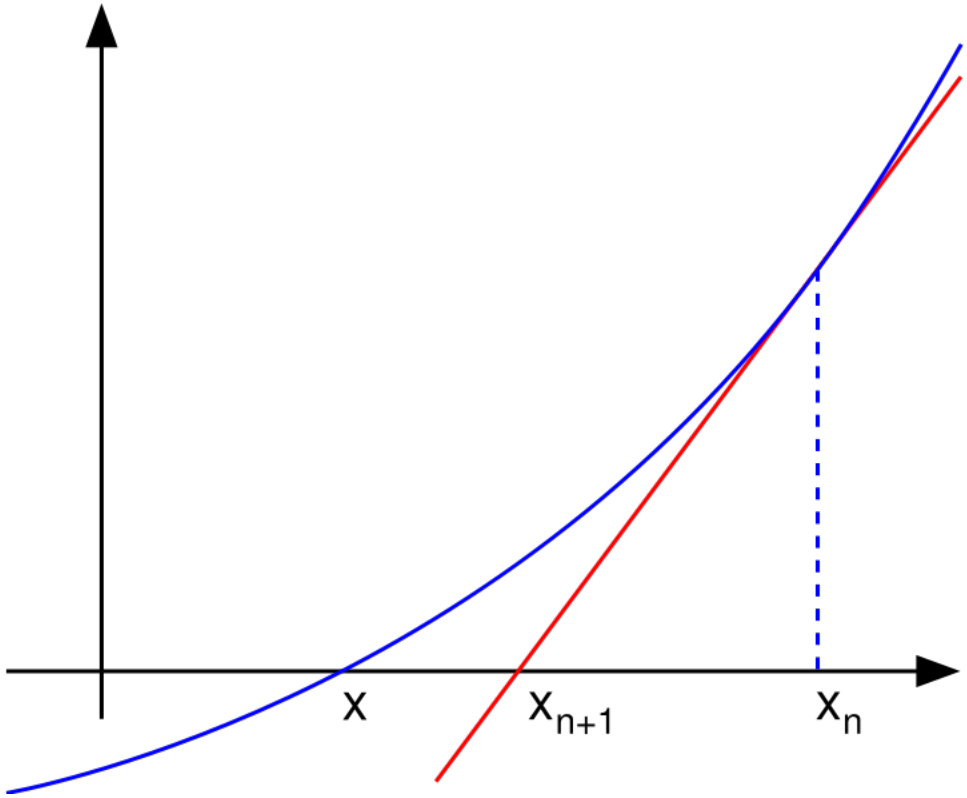
- Syntax

```
while (condition) {
    statements
}
```

  - The **condition** is evaluated, and if it evaluates to FALSE, nothing more is done;
  - If it evaluates to TRUE:
    - the statements are run
    - **condition** is evaluated again, and the process is repeated

Example: Newton's method for root finding

- Find the root of an algebraic equation: *f(x)=0*



Example: Newton's method for root finding

- if *f(x)* has derivative *f'(x)*, then the following iterations will converge to a root of *f(x) = 0* if started close enough to the root:

$$x_0 = \text{initial guess}$$

$$f'(x_n) = \frac{f(x_n)}{x_n - x_{n+1}}$$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

$$f(x) = x^3 + 2x^2 - 7$$

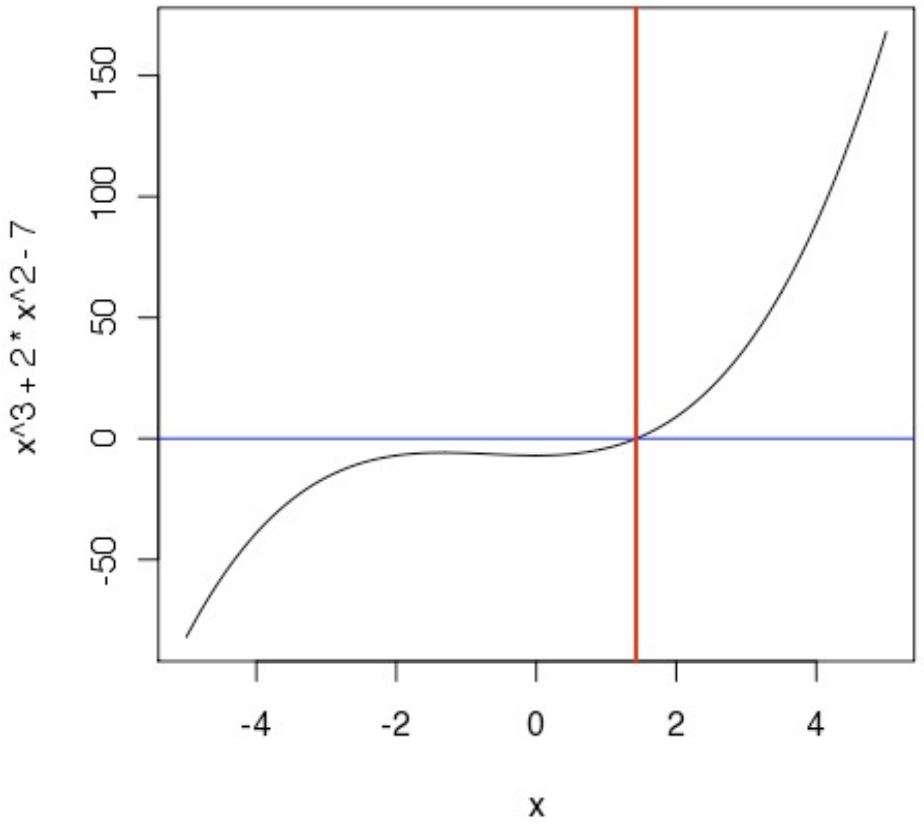
$$x_{n+1} = x_n - \frac{x_n^3 + 2x_n^2 - 7}{3x_n^2 + 4x_n}$$

Example: Newton's method for root finding

```
x0 <- 1

x <- x0
f <- x^3 + 2 * x^2 - 7
tolerance <- 0.000001

while (abs(f) > tolerance) {
  f.prime <- 3 * x^2 + 4 * x
  x <- x - f / f.prime
  f <- x^3 + 2 * x^2 - 7
  print(x)
}
```



Example: Newton's method for root finding

```
# start values as a vector
x0 <- seq(-3, 3, length = 30)

x <- x0
f <- x^3 + 2 * x^2 - 7
tolerance <- 0.000001

while (max(abs(f)) > tolerance) {   # note the change
  f.prime <- 3 * x^2 + 4 * x
  x <- x - f / f.prime
  f <- x^3 + 2 * x^2 - 7
  print(x)
}
```

repeat **loop and** break

- not to put the condition test at the top of the loop
- Syntax

```
repeat {
   statements
   if (condition)
      break
   statements
}
```

  - **break** causes the loop to terminate immediately
  - **break** can also be used in **for()** and **while()** loops.
    - Such use is not recommended for coding clarity, but sometimes can be handy.

Example: Newton's method for root finding

```
x <- 1
tolerance <- 0.000001
repeat {
  f <- x^3 + 2 * x^2 - 7    # only occurs once
  if (abs(f) < tolerance)
    break
  f.prime <- 3 * x^2 + 4 * x
  x <- x - f / f.prime
  print(x)
}
```

Improving speed performance by avoiding loops

- Looping over very large data sets can become slow in R.
- Overcome it by avoiding loops over the data intensive dimension in an object.
  - performing vector-to-vector or matrix-to-matrix computations which run often over 100 times faster than the corresponding for() or apply() loops in R
  - make use of the existing speed-optimized R functions (*e.g.:* **rowSums, rowMeans, table, tabulate**)

Speed comparison example

```
myMA <- matrix(rnorm(1000000), 100000, 10,
    dimnames=list(1:100000, paste("C", 1:10, sep="")))
dim(myMA)
head(myMA)

system.time(myMAmean <- apply(myMA, 1, mean))
myMAmean[1:4]

system.time(myMAmean <- rowMeans(myMA))
myMAmean[1:4]

# The rowMeans approach is over 200 times faster than the apply loop
```

Speed comparison example

```
system.time(myMAsd <- apply(myMA, 1, sd))
myMAsd[1:4]

system.time(myMAsd <- sqrt((rowSums((myMA-rowMeans(myMA))^2)) /
(length(myMA[1,])-1) ))
myMAsd[1:4]

# The vector-based approach in the last step is over 100 times
# faster than the apply loop.
```

Managing complexity through R functions

Most practical programs are much longer than the R examples so far.

- Difficult to keep the all the details in our heads at once.

- Need ways to reduce the complexity.

Managing complexity through R functions

Functions are **self-contained units** with a well-defined purpose.

- Take inputs
- Do calculations, print results, draw graphs, call other functions
- Produce outputs and return values
- If inputs and outputs are well-defined, we can be reasonably sure whether the function works or not, and make it work
- Once it works, we can move on to the next problem

Benefits of using functions

- Reduces a large problem to several smaller ones.
- Make code less complicated and repetitive
- Can reuse a function in different tasks
- Can be flexibly modified to use parameters to augment the behavior of a function

Example: a function to generate *n* Fibonacci numbers

Syntax:

```
function.name <- function(parameter1, parameter2,...)
   {
   function.body
}
```

Example:

```
FibonacciNum <- function(n) {
   Fib <- numeric(n) # allocate variable
   Fib[1] <- Fib[2] <- 1

   for (i in 3:n) {
       Fib[i] <- Fib[i - 2] + Fib[i - 1]
   }
   Fib       # the last expression is returned
}
```

Function definition structure

1. The word **function**

2. A pair of parentheses **( )**, which enclose the argument list. The list may be empty

3. A single statement, or a sequence of statements enclosed in curly braces **{ }**

Functions

- When R executes a function definition, it produces a **function object** with three parts:
  - Header (function name, argument list)
  - Body
  - A reference to the environment in which the function is defined

- In R, functions are objects that can be manipulated like other common objects such as vectors, matrices, and lists
  - can be used as function arguments: **apply(mat, 1, mean)**

Header

- Choose informative function names to facilitate code writing
  - describe the action of the function
  - easy to remember
  - not misleading or confused with other function names
  - similarly choose argument names

- Arguments can take default values
  - if no value is specified for an argument, the default will be used

```
FibDefault <- function(n = 10) {
    FibonacciNum(n)
}
```

Body

- One or more statements in { }
- Specify what computations are to be carried out by the function
- **`return(value)`** to specify the output value of the function
  - can be in the middle of the body
  - all R functions produce a **single** output
    - use list to combine multiple values
  - if no return() is used, the value of the **last statement executed** is returned

The environment of the function

```
PrimeSieve <- function(n) {
# return all prime numbers up to n

  if (n >= 2) {
    sieve <- seq(2, n)   # sieve initially contains 2:n
    primes <- c()        # empty vector

    for (i in seq(2, n)) {

      if (any(sieve == i)) {    # sieve contains i
        primes <- c(primes, i)  # add i to prime list
        sieve <- c(sieve[(sieve %% i) != 0])  # multiples of i are removed
      }
    }
    return(primes)
  } else {
    print("Input value of n should be >= 2")
  }
}
```

The environment of the function

```
PrimeSieve2 <- function(n) {
# return all prime numbers up to n

  if (n >= 2) {
    noMultiples <- function(k) {
      ### function within a function; make removal clearer
      sieve[(sieve %% k) != 0]

    }

    sieve <- seq(2, n)   # sieve initially contains 2:n
    primes <- c()        # empty vector

    for (i in seq(2, n)) {

      if (any(sieve == i)) {    # sieve contains i
        primes <- c(primes, i)  # add i to prime list

        #sieve <- c(sieve[(sieve %% i) != 0])  # multiples of i are removed
        sieve <- noMultiples(i)   ### call the new function instead
      }
    }
    return(primes)
  } else {
    print("Input value of n should be >= 2")
  }
}
```

The environment of the function

- A reference to the environment in which the function is defined

- Referring to **PrimeSieve2()**
  - locally defined variables within the function: **n, sieve, primes, i**
  - **seq, c, any, return, print** are not defined here: they are part of the R environment where PrimeSieve() is defined
  - a function **noMultiples()** defined within PrimeSieve2()
    - This function's environment includes **n, seive, primes, i**
    - It has local variable **k**

Scope of variables

- The **scope** of a variable tells where the variable would be recognized.
  - Variables defined at the console have **global scope** – visible in user-defined functions.
  - Variables defined within functions have **local scope**, so they are only recognized within the function
    - no need to worry that the local variables will be changed by other parts of the program
  - A variable could be created with the same name in a different function but there is no risk of a clash
    - R packages are collections of functions and data, and have their own scope (“namespace”)

Scope of variables

```
x <- 1:10   # global variable

f <- function() {
  x <- 1
  g()           # g will have no effect on local x
  # print(g())
  return(x)
}

g <- function() {
  x.g <- 100  # local variable
  x <- 2   # changes local x in g, not the one in f
  return(x)
}
```

General programming guidelines

Steps to write a program to solve a problem

1. Understand the problem
2. Work out a general idea how to solve it
3. Translate your general idea into a detailed implementation
4. Check: does it work? Is it good enough?
    - If yes, done.
    - If no, go back to step 2.

Examples: Sort a vector of integers into increasing order

1. Understand the problem

   – A good way is to try a specific and simple case (but not too trivial)

     • consider sorting 3, 5, 24, 6, 2, 4, 13, 1

   – We will try a function **my.sort()**

     ```
     x <- c(3, 5, 24, 6, 2, 4, 13, 1)
     my.sort(x)
     ```

     The output should be: 1, 2, 3, 4, 5, 6, 13, 24

2. Work out a general idea

- One idea (selection sort)
  - Find the smallest number
  - Swap it with the value in the 1st position
  - Repeat the steps for the remainder of the list (starting at the 2nd position and advancing each time)

- Alternative idea (bubble sort)
  - Compare successive pairs of values, starting at the beginning of the vector, and running through to the end.
  - Swap pairs if they are out of order
    - Try it on **2, 1, 4, 3, 0**, does it work?
    - End up with 1, 2, 3, 0, 4
      - The largest value always lands at the end of the new vector
      - Repeat the procedure for all but the last value, and so on

3. Detailed implementation

- Address specific coding questions
  - e.g. how to successive swap adjacent pairs?

```
save <- x[i]
x[i] <- x[i + 1]
x[i + 1] <- save
```

- Write the code

```r
my.sort <- function(x) {

  # x is the initial vector and will
  # be modified to from the output

  for (last in length(x):2) {
    # put the largest value at the end
    # first is compared with last

    for (first in 1:(last - 1)) {
      if (x[first] > x[first + 1]) { # swap the pair
        save <- x[first]
        x[first] <- x[first + 1]
        x[first + 1] <- save
      }
    }
  }

  return(x)
}
```

4. Check

Begin testing on simple examples to identify obvious bugs.

```
my.sort(c(2,1))

x <- c(3, 5, 24, 6, 2, 4, 13, 1)
my.sort(x)

my.sort(1)
```

```r
my.sort <- function(x) {

  # x is the initial vector and will
  # be modified to from the output

  if (length(x) < 2)   ### handle special case
    return(x)

  for (last in length(x):2) {
    # put the largest value at the end
    # first is compared with last

    for (first in 1:(last - 1)) {
      if (x[first] > x[first + 1]) { # swap the pair
        save <- x[first]
        x[first] <- x[first + 1]
        x[first + 1] <- save
      }
    }
  }

  return(x)
}
```

Top-down design to work out the detailed implementation

Break down a large problem into smaller pieces that are easier to solve.

- similar to outlining an essay before filling in the details

1. Write out the whole program in a small number (1-5) of steps, in plain language

2. Expand each step into a small number of steps,
   - write the code if the task is doable

3. Keep going until you have a program

Code indentation

- **Always intent** the code when a subsection starts
  - code chunk after if(), for(), while()
  - align a finishing **}** with the starting column of the paired **{**
    - easy to visually check pairing of { }
  - see the PrimeSieve() example

Proofread code before running!

- avoid errors due to typos
  - e.g. use **num.people** first, but **num.People** later
  - don’t use **i** and **j** in a double loop
- compiling errors are easy to identify, but logical errors are hard: avoid logical errors up front
- save debug time: debugging is like reverse-engineering
  - Once you run the code and there are errors, the error messages often appear irrelevant and don't point you to the problematic code.
- Relevance to R: R doesn’t require you to declare a variable name before using it

Steps to find and fix bugs

1. Recognize that a bug exists
2. Make the bug reproducible
3. Identify the cause of the bug
4. Fix the errors and test
5. Look for similar errors

Recognize bugs

- Break up your program into simple, self-contained functions.
  - Document inputs and outputs.
  - Test each function to make sure it works, before calling it in another function

- Test both simple and non-trivial inputs.
  - Try a vector of length 0
  - Try very large or very small values

- When error occurs only for certain inputs
  - Look for “edge cases”: boundary between legal and illegal inputs

Performing Simple Linear Regression

- **Problem**
  - You have two vectors, x and y, that hold paired observations: (x_1, y_1), (x_2, y_2), ..., (x_n, y_n). You believe there is a linear relationship between x and y, and you want to create a regression model of the relationship:

    $y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$

- **Solution**
  - The **lm** function performs a linear regression and reports the coefficients:

    ```
    > lm(y ~ x)
    ```

Performing Simple Linear Regression

```
> head(dfrm)
          x          y
1 0.04781401   5.406651
2 1.90857986  19.941568
3 2.79987246  23.922613
...

> lm(y ~ x, data=dfrm)
Call:
lm(formula = y ~ x, data = dfrm)
Coefficients:
(Intercept)            x
      17.72         3.25
```

Performing Multiple Linear Regression

- **Problem**
  - You have several predictor variables (e.g., u, v, and w) and a response variable y. You believe there is a linear relationship between the predictors and the response, and you want to perform a linear regression on the data:

    $y_i = \beta_0 + \beta_1 u_i + \beta_2 v_i + \beta_3 w_i + \varepsilon_i$

- **Solution**
  - R uses the **lm** function for both simple and multiple linear regression. You simply add more variables to the righthand side of the model formula. The output then shows the coefficients of the fitted model:

    ```
    > lm(y ~ u + v + w)
    ```

Performing Multiple Linear Regression

```
> head(dfrm)
        y           u           v           w
1  6.584519  0.79939065  2.7971413  4.366557
2  6.425215 -2.31338537  2.7836201  4.515084
3  7.830578  1.71736899  2.7570401  3.865557
...

> lm(y ~ u + v + w, data=dfrm)
Call:
lm(formula = y ~ u + v + w, data = dfrm)
Coefficients:
(Intercept)   u         v         w
1.4222        1.0359    0.9217    0.7261
```

Getting Regression Statistics

- **Problem**
  - You want the critical statistics and information regarding your regression.
- **Solution**
  - Save the regression model in a variable, say m:

    ```
    > m <- lm(y ~ u + v + w)
    ```
  - Then use functions to extract regression statistics and information from the model:
    - anova(m)
      - ANOVA table
    - coefficients(m) or coef(m)
      - Model coefficients
    - confint(m)
      - Confidence intervals for the regression coefficients

Getting Regression Statistics

- deviance(m)
  - Residual sum of squares
- effects(m)
  - Vector of orthogonal effects
- fitted(m)
  - Vector of fitted *y* values
- residuals(m) or resid(m)
  - Model residuals
- summary(m)
  - Key statistics, such as R^2, the F statistic, the residual standard error
- vcov(m)
  - Variance–covariance matrix of the main parameters

Getting Regression Statistics

```
> m <- lm(y ~ u + v + w)
> summary(m)
Call:
lm(formula = y ~ u + v + w)
Residuals:
Min                 1Q        Median      3Q        Max
-3.3965      -0.9472  -0.4708     1.3730    3.1283
Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.4222    1.4036     1.013   0.32029
u                1.0359    0.2811     3.685   0.00106 **
v                0.9217    0.3787     2.434   0.02211 *
w                 0.7261   0.3652     1.988   0.05744 .
---
Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
Residual standard error: 1.625 on 26 degrees of freedom
Multiple R-squared: 0.4981, Adjusted R-squared: 0.4402
F-statistic: 8.603 on 3 and 26 DF, p-value: 0.0003915
```

Getting Regression Statistics

```
> coef(m)
(Intercept)           u           v           w
  1.4222050   1.0358725   0.9217432   0.7260653
> confint(m)
                2.5 %     97.5 %
(Intercept) -1.46302727  4.307437
u            0.45805053  1.613694
v            0.14332834  1.700158
w           -0.02466125  1.476792
> resid(m)
          1           2           3           4           5           6
-1.41440465  1.55535335 -0.71853222 -2.22308948 -0.60201283 -0.96217874
          7           8           9          10          11          12
-1.52877080  0.12587924 -0.03313637  0.34017869  1.28200521 -0.90242817
         13          14          15          16          17          18
 2.04481731  1.13630451 -1.19766679 -0.60210494  1.79964497  1.25941264
... ...
```

Getting Regression Statistics

```
> deviance(m)
[1] 68.69616

> anova(m)
Analysis of Variance Table
Response: y
          Df  Sum Sq  Mean Sq F value   Pr(>F)
u          1  27.916  27.9165  10.5658 0.003178 **
v          1  29.830  29.8299  11.2900 0.002416 **
w          1  10.442  10.4423  3.9522  0.057436 .
Residuals 26  68.696  2.6422
---
Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Finding the Best Power Transformation

- **Problem**
  - You want to improve your linear model by applying a power transformation to the response variable.

- **Solution**
  - Use the Box–Cox procedure, which is implemented by the **boxcox** function of the MASS package. The procedure will identify a power, λ, such that transforming y into y^{λ} will improve the fit of your model:

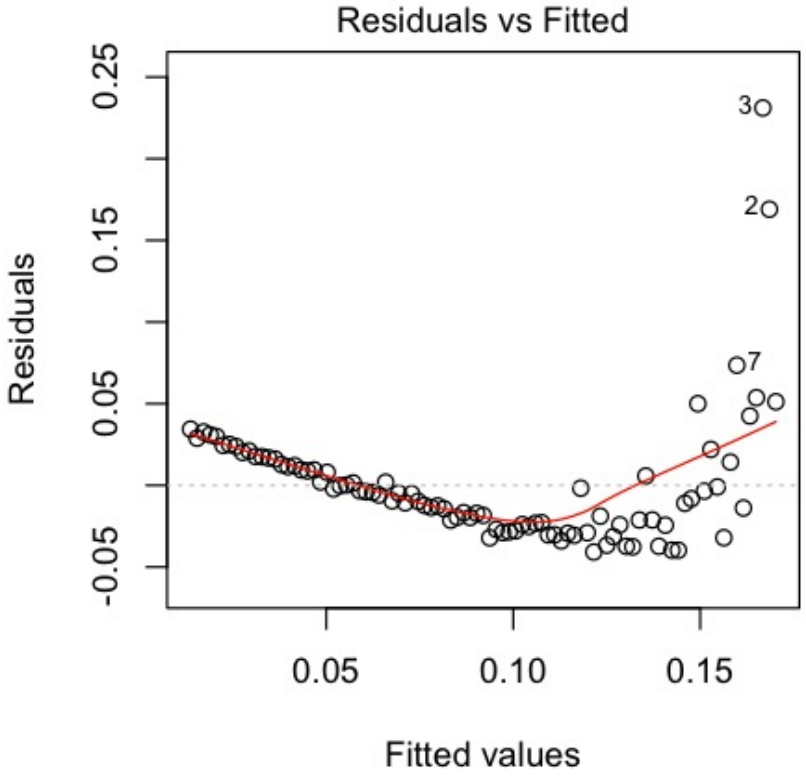
```
> library(MASS)
> m <- lm(y ~ x)
> boxcox(m)
```

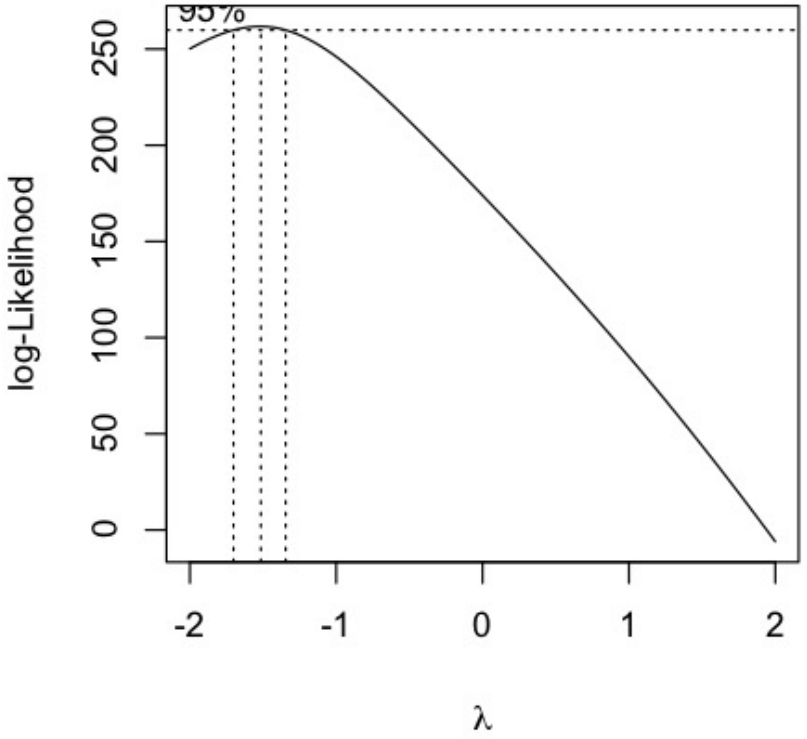
Finding the Best Power Transformation

```
> x <- 10:100
> eps <- rnorm(length(x), sd=5)
> y <- (x + eps)^(-1/1.5)
> m <- lm(y ~ x)
> summary(m)

Call:
lm(formula = y ~ x)

Residuals:
      Min        1Q    Median        3Q       Max
-0.040793 -0.024069 -0.005941  0.013467  0.231081

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.1876515  0.0095772   19.59   <2e-16 ***
x           -0.0017395  0.0001571  -11.07   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.03937 on 89 degrees of freedom
Multiple R-squared: 0.5793,      Adjusted R-squared: 0.5746
F-statistic: 122.6 on 1 and 89 DF,  p-value: < 2.2e-16
```

Finding the Best Power Transformation

```
> plot(m, which=1)
> library(MASS)
> bc <- boxcox(m)
> which.max(bc$y)
[1] 13
> lambda <- bc$x[which.max(bc$y)]
> lambda
[1] -1.515152
```





Finding the Best Power Transformation

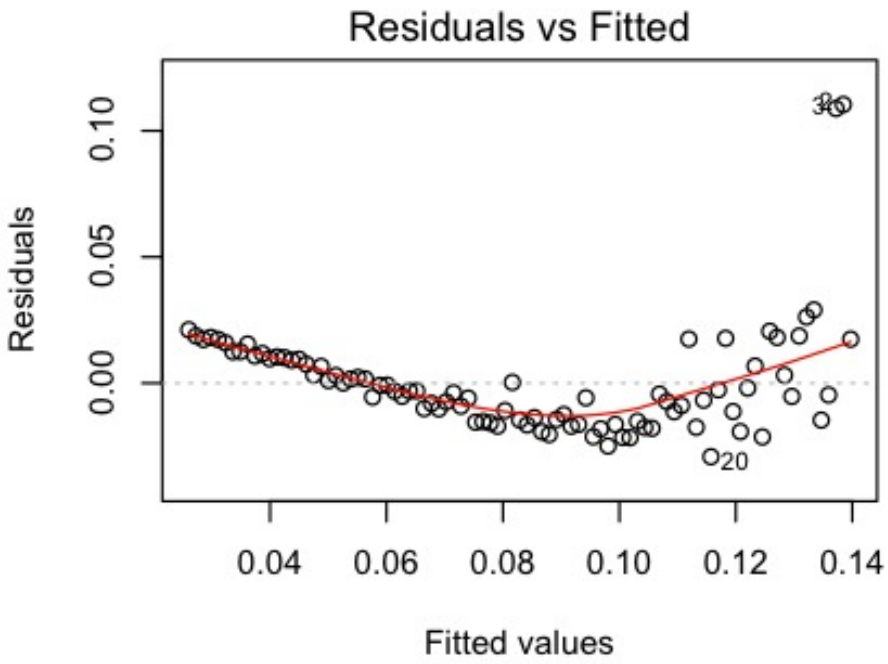
```
> z <- y^lambda
> m2 <- lm(z ~ x)
> summary(m2)

Call:
lm(formula = z ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-13.5091  -3.2048  -0.5666   3.7794  13.8832

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -3.28064    1.25003  -2.624   0.0102 *
x            1.07079    0.02051  52.211   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 5.139 on 89 degrees of freedom
Multiple R-squared: 0.9684,      Adjusted R-squared: 0.968
F-statistic:  2726 on 1 and 89 DF,  p-value: < 2.2e-16
```

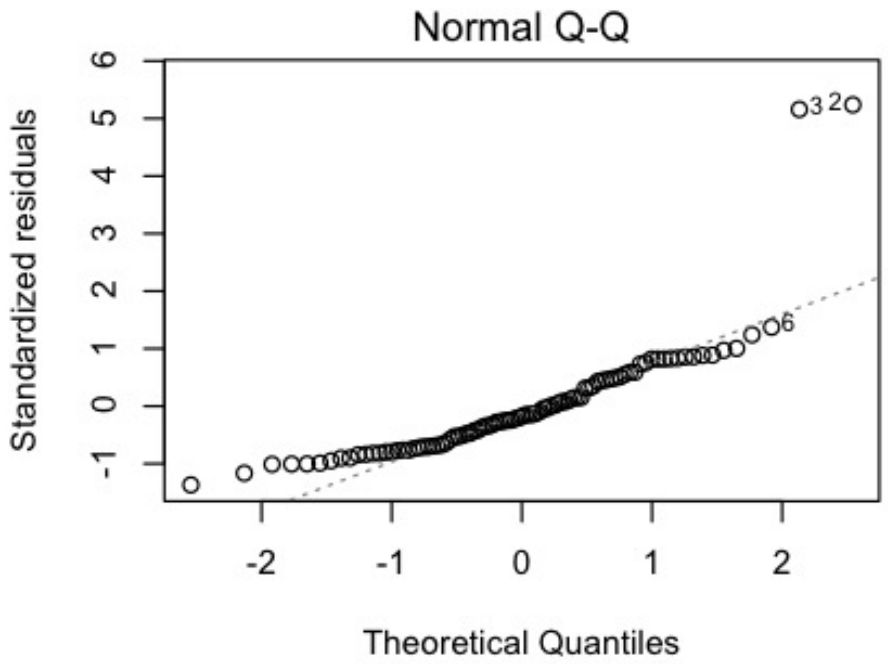
Diagnosing a Linear Regression

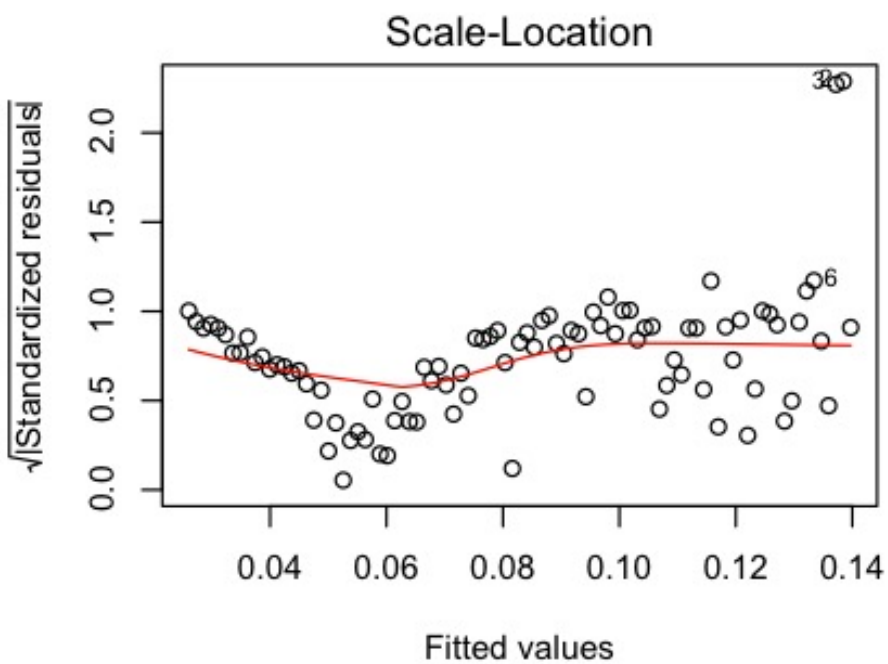
- **Problem**
  - You have performed a linear regression. Now you want to verify the model's quality by running diagnostic checks.

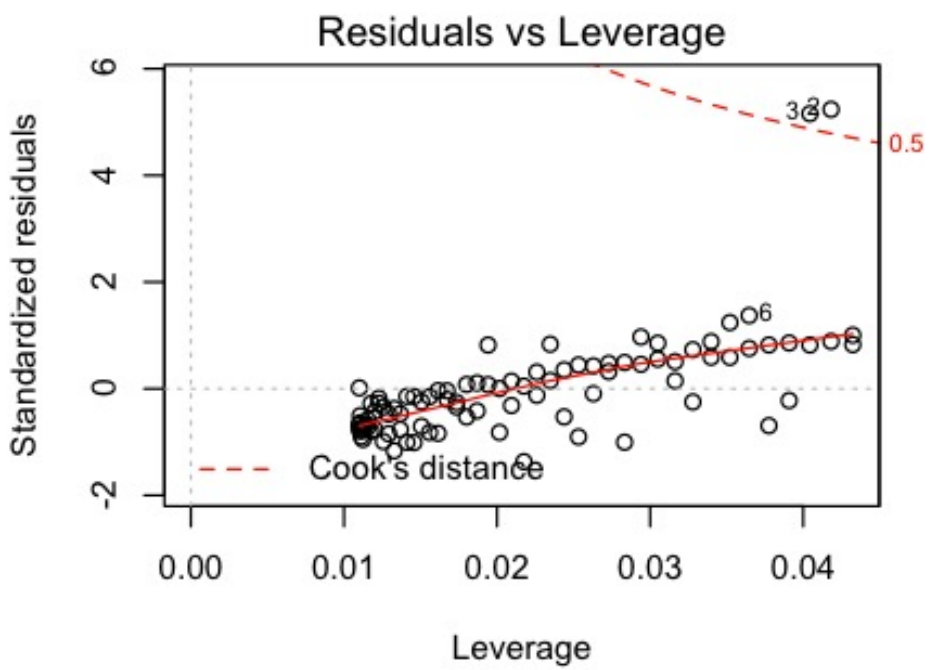
- **Solution**
  - You have performed a linear regression. Now you want to verify the model's quality by running diagnostic checks:

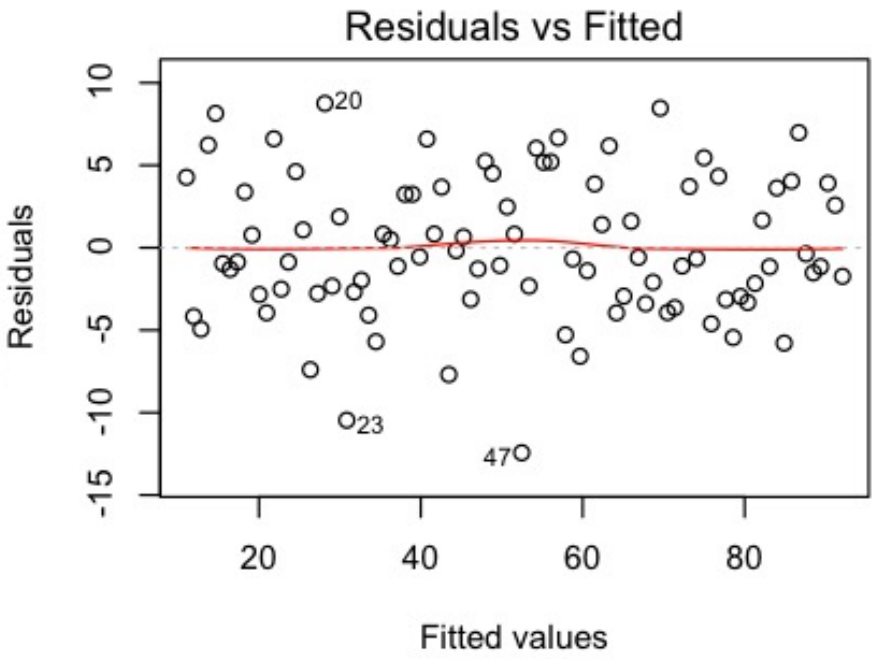
```
> m <- lm(y ~ x)
> plot(m)
```

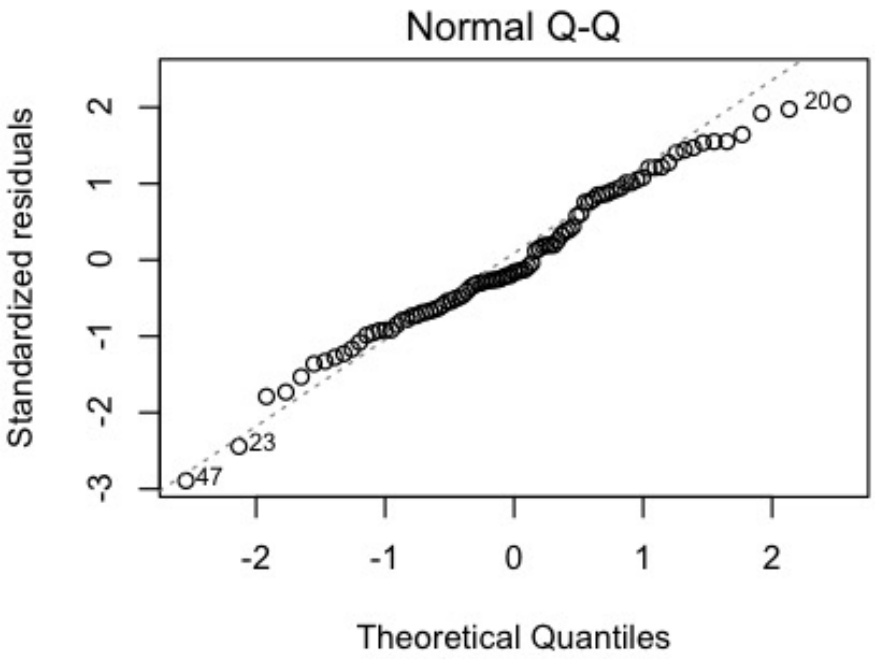

Residuals vs Fitted
Residuals
Fitted values
0.00
0.05
0.10
0.04
0.06
0.08
0.10
0.12
0.14
3
20

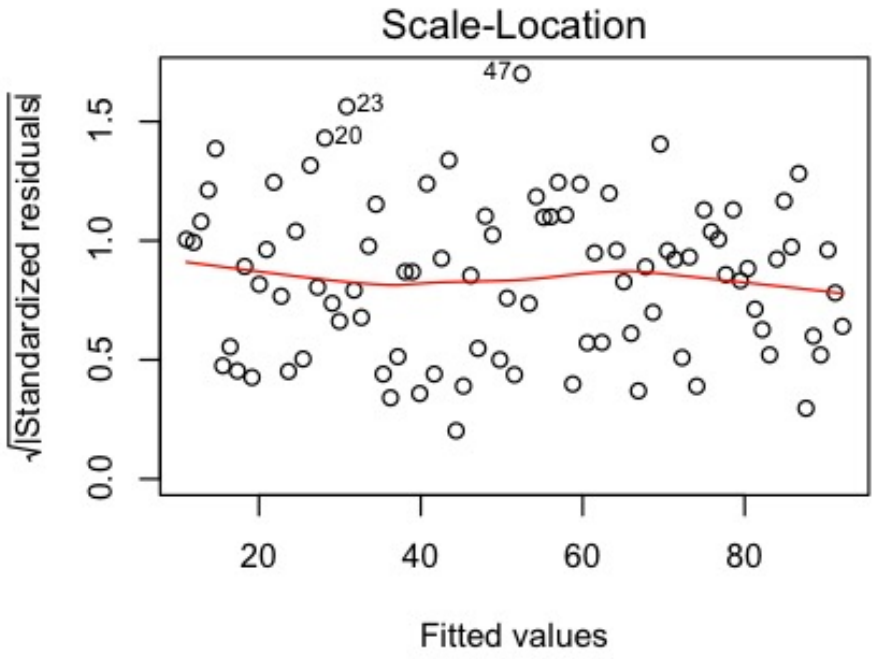

Normal Q-Q
Standardized residuals
Theoretical Quantiles
-1
0
1
2
3
4
5
6
-2
-1
0
1
2
3
2
6

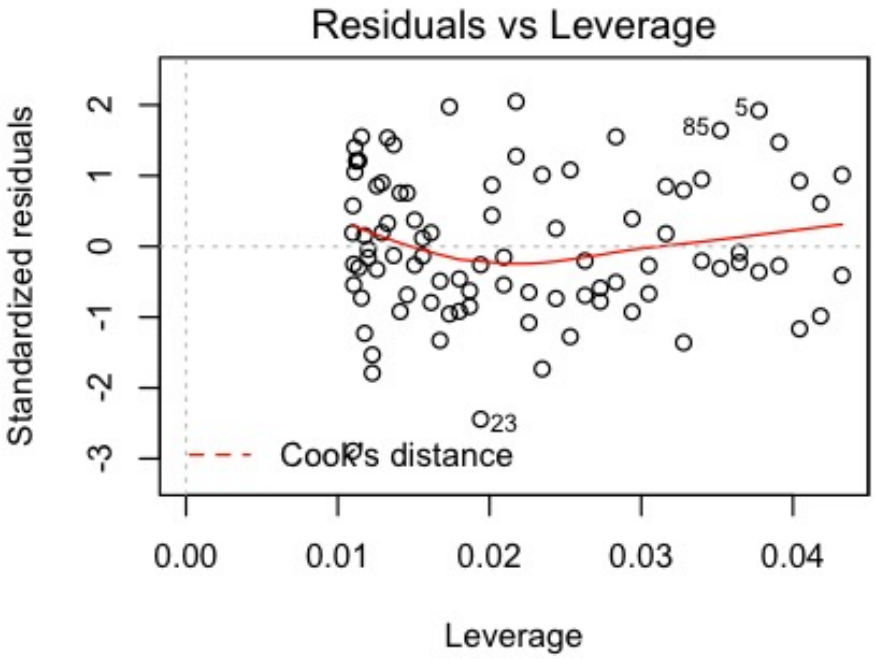

Scale-Location
√|Standardized residuals|
Fitted values
0.0
0.5
1.0
1.5
2.0
0.04
0.06
0.08
0.10
0.12
0.14
3
6


Residuals vs Leverage
Standardized residuals
Leverage
Cook's distance
0.5
-2
0
2
4
6
0.00
0.01
0.02
0.03
0.04
3
2
6


Residuals vs Fitted
Residuals
Fitted values
20
23
47


Normal Q-Q
Standardized residuals
Theoretical Quantiles
20
23
47


Scale-Location
√|Standardized residuals|
Fitted values
47
23
20


Residuals vs Leverage
Standardized residuals
Leverage
Cook's distance
85
5
23

Diagnosing a Linear Regression

```
> library(car)
> outlierTest(m)
  rstudent      unadjusted p-value       Bonferonni p
2   6.261697           1.356e-08         1.2340e-06
3   6.125430           2.476e-08         2.2531e-06
> outlierTest(m2)

No Studentized residuals with Bonferonni p < 0.05
Largest |rstudent|:
    rstudent    unadjusted p-value       Bonferonni p
47 -3.021435          0.0032938          0.29973
```

Predicting New Values

- **Problem**
  - You want to predict new values from your regression model.

- **Solution**
  - Save the predictor data in a data frame. Use the **predict** function, setting the newdata parameter to the data frame:

```
> m <- lm(y ~ u + v + w)
> preds <- data.frame(u=3.1, v=4.0, w=5.5)
> predict(m, newdata=preds)
```

Forming Prediction Intervals

- **Problem**
  - You are making predictions using a linear regression model. You want to know the prediction intervals: the range of the distribution of the prediction.

- **Solution**
  - Use the **predict** function and specify interval="prediction":

```
> preds <- data.frame(u=c(3.0, 3.1), v=c(3.9, 4.0), w=c(5.3, 5.5))
> predict(m, newdata=preds, interval="prediction")
       fit      lwr      upr
1 11.97277 7.999848 15.94569
2 12.31374 8.272327 16.35516
```

Performing One-Way ANOVA

- **Problem**
  - Your data is divided into groups, and the groups are normally distributed. You want to know if the groups have significantly different means.

- **Solution**
  - Use a factor to define the groups. Then apply the **oneway.test** function:

```
> oneway.test(x ~ f)       # default: var.equal=FALSE
> m <- aov(x ~ f)   # always assumes equal variances
> summary(m)
```

Performing One-Way ANOVA

```
> oneway.test(iris$Sepal.Length ~ iris$Species)
        One-way analysis of means (not assuming equal variances)

data:  iris$Sepal.Length and iris$Species
F = 138.91, num df = 2.000, denom df = 92.211, p-value < 2.2e-16
```

Performing Robust ANOVA

- **Problem**
  - Your data is divided into groups. The groups are not normally distributed, but their distributions have similar shapes. You want to perform a test similar to ANOVA - you want to know if the group medians are significantly different.

- **Solution**
  - Create a factor that defines the groups of your data. Use the **kruskal.test** function, which implements the Kruskal-Wallis test. Unlike the ANOVA test, this test does not depend upon the normality of the data:

```
> kruskal.test(x ~ f)
```

Performing Robust ANOVA

```
> print(midterm)
[1] 0.8182 0.6818 0.5114 0.6705 0.6818 0.9545 0.9091 0.9659 1.0000 0.7273
[11] 0.7727 0.7273 0.6364 0.6250 0.6932 0.6932 0.0000 0.8750 0.8068 0.9432
… …
> print(hw)
[1] 4 4 2 3 4 4 4 3 4 4 3 3 1 4 4 3 3 4 3 4 4 4 3 2 4 4 4 4 3 2 3 4 4 0 4 4 4 4
[39] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 2 4 4 4 4 4 1 4 4 4 4 4 4 4 4 2 4 4
[77] 3 4 2 2 4 4 4 4 1 3 0 4 4 0 4 3 4 4
Levels: 0 1 2 3 4
> kruskal.test(midterm ~ hw)
        Kruskal-Wallis rank sum test
data: midterm by hw
Kruskal–Wallis chi-squared = 25.6813, df = 4, p-value = 3.669e-05
```

Comparing Models by Using ANOVA

- **Problem**
  - You have two models of the same data, and you want to know whether they produce different results.

- **Solution**
  - The **anova** function can compare two models and report if they are significantly different (m1 and m2 are both model objects returned by **lm**):

```
> anova(m1, m2)
```

Comparing Models by Using ANOVA

```
> m1 <- lm(y ~ u)
> m2 <- lm(y ~ u + v)
> anova(m1, m2)
Analysis of Variance Table
Model 1: y ~ u
Model 2: y ~ u + v
  Res.Df    RSS  Df  Sum of Sq      F    Pr(>F)
1     28 108.968
2     27  79.138   1     29.83 10.177 0.003587 **
---
Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Performing Principal Component Analysis

- **Problem**
  - You want to identify the principal components of a multivariable dataset.

- **Solution**
  - Use the **prcomp** function. The first argument is a formula whose righthand side is the set of variables, separated by plus signs (+). The lefthand side is empty:

```
> r <- prcomp( ~ x + y + z)
> summary(r)
```

Performing Principal Component Analysis

```
> r <- prcomp( ~ x + y)
> print(r)
Standard deviations:
[1] 0.3724471 0.1306414
Rotation:
      PC1        PC2
x -0.5312726 -0.8472010
y -0.8472010  0.5312726
> summary(r)
Importance of components:
                          PC1   PC2
Standard deviation      0.372 0.131
Proportion of Variance  0.890 0.110
Cumulative Proportion   0.890 1.000
```

Finding Clusters in Your Data

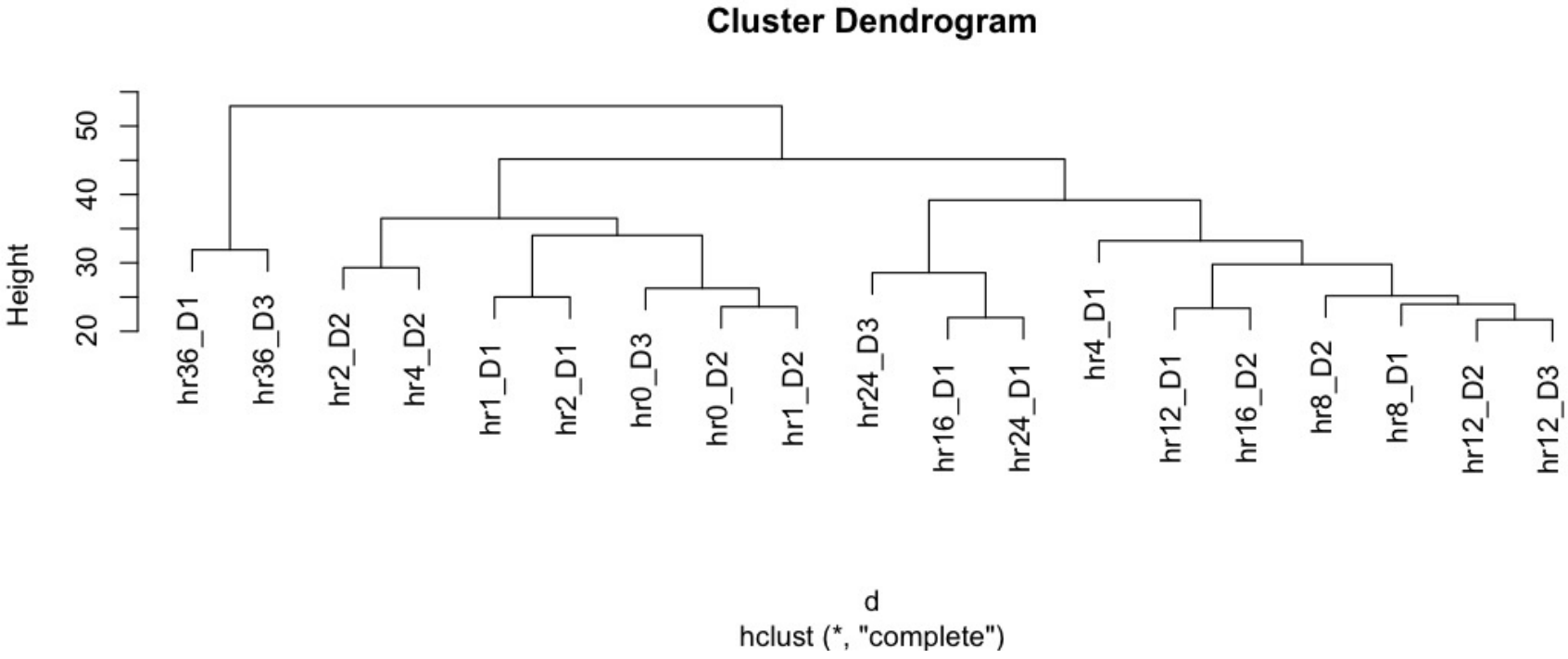
- **Problem**
  - You believe your data contains clusters: groups of points that are “near” each other. You want to identify those clusters.

- **Solution**
  - Your dataset, x, can be a vector, data frame, or matrix. Assume that n is the number of clusters you desire:

```
> d <- dist(x)
> hc <- hclust(d)
> clust <- cutree(hc, k=n)
```

Finding Clusters in Your Data

```
> Hirsch <- read.table("Hirsch_Cancer_Cell.tab", row.names=1, header=TRUE)
> d <- dist(t(as.matrix(Hirsch)))
> hc <- hclust(d)
> clust <- cutree(hc, k=3)
> plot(hc)
```



Predicting a Binary-Valued Variable (Logistic Regression)

- **Problem**
  - You want to perform logistic regression, a regression model that predicts the probability of a binary event occurring.

- **Solution**
  - Call the **glm** function with family=binomial to perform logistic regression. The result is a model object:

    ```
    > m <- glm(b ~ x1 + x2 + x3, family=binomial)
    ```

  - Use the model object, m, and the **predict** function to predict a probability from new data:

    ```
    > dfrm <- data.frame(x1=value, x2=value, x3=value)
    > predict(m, type="response", newdata=dfrm)
    ```

Predicting a Binary-Valued Variable (Logistic Regression)

```
> data(pima, package="faraway")
> b <- factor(pima$test)
> m <- glm(b ~ diastolic + bmi, family=binomial, data=pima)
> summary(m)
> newdata <- data.frame(bmi=32.0, diastolic=75)
> predict(m, type="response", newdata=newdata)
        1
0.3318973
```

Summary

- R Cookbook
  - Chapter 11. Linear Regression and ANOVA
  - Chapter 13. Beyond Basic Numerics and Statistics