An Overview of the R Programming Environment

Deepayan Sarkar

Software for Statistics

- Computing software is essential for modern statistics
  - Large datasets
  - Visualization
  - Simulation
  - Iterative methods
- Many softwares are available, but we will focus on R
- Why R?
  - Available as Free / Open Source Software
  - Very popular (both academia and industry)
  - Easy to try out on your own
- Also because I know R better than other languages (Python and Julia are other good alternatives)

Outline

- Installing R
- Basics of using R
- Some examples
- Fitting linear models in R

Installing R

- R is most commonly used as a REPL (Read-Eval-Print-Loop)
  - When it is started, R Waits for user input
  - Evaluates and prints result
  - Waits for more input
- There are several different *interfaces* to do this
- R itself works on many platforms (Windows, Mac, UNIX, Linux)
- Some interfaces are platform-specific, some work on most
- R and the interface may need to be installed separately
- Go to https://cran.r-project.org/ (or choose a mirror first)
- Follow instructions depending on your platform (probably Windows)
- This will install R, as well as a default graphical interface on Windows and Mac

- I will recommend a different interface called R Studio that needs to be installed separately
- I personally use yet another interface called ESS which works with a general purpose editor called Emacs (download link for Windows)

Running R

- Once installed, you can start the appropriate interface (or R directly) to get something like this:

```
R Under development (unstable) (2018-05-05 r74699) -- "Unsuffered Consequences"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

Loading required package: utils
>
```

- The `>` represents a *prompt* indicating that R is waiting for input.
- The difficult part is to learn what to do next

Basic usage

The R REPL essentially works like a calculator

```r
34 * 23
```

```
[1] 782
```

```r
27 / 7
```

```
[1] 3.857143
```

```r
exp(2)
```

```
[1] 7.389056
```

```r
2^10
```

```
[1] 1024
```

R has standard mathematical functions

```r
sqrt(5 * 125)
```

```
[1] 25
```

```r
log(120)
```

```
[1] 4.787492
```

```r
factorial(10)
```

```
[1] 3628800
```

```r
log(factorial(10))
```

```
[1] 15.10441
```

```r
choose(15, 5)
```

```
[1] 3003
```

```r
factorial(15) / (factorial(10) * factorial(5))
```

```
[1] 3003
```

```r
choose(1500, 2)
```

```
[1] 1124250
```

```r
factorial(1500) / (factorial(1498) * factorial(2))
```

```
[1] NaN
```

R supports variables

```r
x <- 2
y <- 10
x^y
```

```
[1] 1024
```

```r
y^x
```

```
[1] 100
```

```r
factorial(y)
```

```
[1] 3628800
```

```r
log(factorial(y), base = x)
```

```
[1] 21.79106
```

R can compute on vectors

```r
N <- 15
x <- seq(0, N)
N
```

```
[1] 15
```

```r
x
```

```
 [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

```r
choose(N, x)
```

```
 [1]    1   15  105  455 1365 3003 5005 6435 6435 5005 3003 1365  455  105   15    1
```

R has functions for statistical calculations

```
p <- 0.25
choose(N, x) * p^x * (1-p)^(N-x)
```

```
 [1] 1.336346e-02 6.681731e-02 1.559070e-01 2.251991e-01 2.251991e-01 1.651460e-01 9.174777e-02 3.93204
 [9] 1.310682e-02 3.398065e-03 6.796131e-04 1.029717e-04 1.144130e-05 8.800998e-07 4.190952e-08 9.31322
```

```
dbinom(x, size = N, prob = p)
```

```
 [1] 1.336346e-02 6.681731e-02 1.559070e-01 2.251991e-01 2.251991e-01 1.651460e-01 9.174777e-02 3.93204
 [9] 1.310682e-02 3.398065e-03 6.796131e-04 1.029717e-04 1.144130e-05 8.800998e-07 4.190952e-08 9.31322
```

R has functions that work on vectors

```
p.x <- dbinom(x, size = N, prob = p)
sum(x * p.x) / sum(p.x)
```

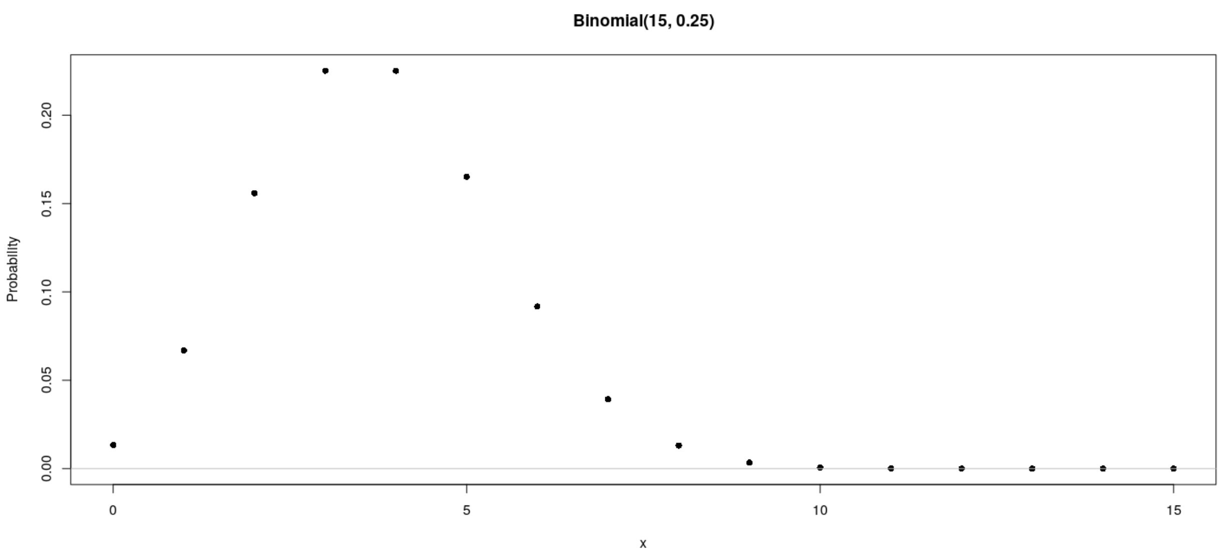
```
[1] 3.75
```

```
N * p
```

```
[1] 3.75
```

R can draw graphs

```
plot(x, p.x, ylab = "Probability", pch = 16)
title(main = sprintf("Binomial(%g, %g)", N, p))
abline(h = 0, col = "grey")
```



R can simulate random variables

```
cards <- as.vector(outer(c("H", "D", "C", "S"), 1:13, paste))
cards
```

```
 [1] "H 1"  "D 1"  "C 1"  "S 1"  "H 2"  "D 2"  "C 2"  "S 2"  "H 3"  "D 3"  "C 3"  "S 3"  "H 4"  "D 4"
[17] "H 5"  "D 5"  "C 5"  "S 5"  "H 6"  "D 6"  "C 6"  "S 6"  "H 7"  "D 7"  "C 7"  "S 7"  "H 8"  "D 8"
[33] "H 9"  "D 9"  "C 9"  "S 9"  "H 10" "D 10" "C 10" "S 10" "H 11" "D 11" "C 11" "S 11" "H 12" "D 12"
```

```
[49] "H 13" "D 13" "C 13" "S 13"
```

```
sample(cards, 13)
```

```
 [1] "S 3"  "C 6"  "D 1"  "S 13" "S 8"  "S 5"  "H 12" "H 3"  "C 10" "C 11" "S 12" "D 3"  "D 12"
```

```
sample(cards, 13)
```

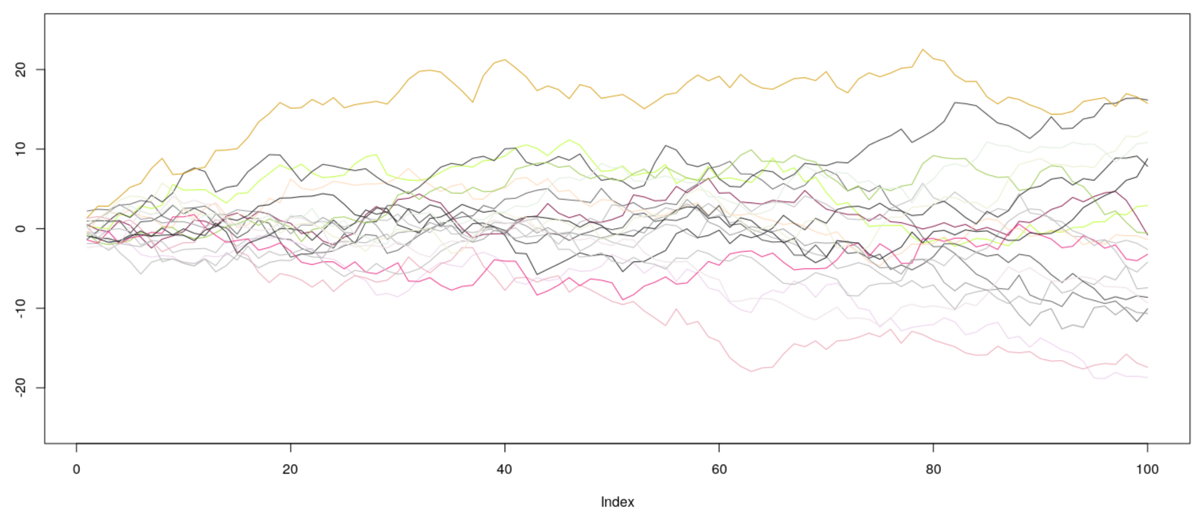
```
 [1] "H 4"  "D 2"  "D 13" "D 12" "C 2"  "D 11" "H 11" "C 4"  "C 10" "S 6"  "C 1"  "D 1"  "H 12"
```

```
z <- rnorm(50, mean = 0, sd = 1)
z
```

```
 [1]  2.3015978047  0.7705667884 -0.0892046760 -0.9865371964  0.1810126429  0.3879747647 -0.2050735532
 [9]  0.7322113434  0.7798364023  0.4075021768  1.1769618106 -0.7462886765  0.2604936430  1.0177229912
[17] -1.1018401259 -0.7095959598  0.0396401117  0.4131531330 -0.2439613859  0.5237437640  1.0509392597
[25] -1.7204412437 -0.2713384125 -0.3736775529 -0.5267496485 -0.7690549718 -1.4408822619 -0.0008896339
[33] -0.4447285252  0.6269810650  0.9813285849  0.6229496408  0.2847310574 -1.0710706779  0.9876113962
[41]  0.1273712433 -0.4372480986 -2.3501188661  1.6769334822  2.4683746362  0.1797619689 -1.3621373802
[49]  1.5429262058  0.1843367986
```

Example: random walk

```
plot(1:100, type = "n", ylim = c(-25, 25), ylab = "")
for (i in 1:20) {
    z <- rnorm(100, mean = 0, sd = 1)
    lines(cumsum(z), col = sample(colors(), 1))
}
```



R is in fact a full programming language

- Variables
- Functions
- Control flow structures
    - For loops, while loops

- If-then-else (branching)

- Distinguishing features
  - Focus on *vectors* and *vectorized operations*
  - Treatment of *functions* at par with other object types
- We will see a few examples to illustrate what I mean by this

Some examples

Before we start, an experiment!

Color combination: Is it **white & gold** or **blue & black** ? Let's count!

Question: What proportion of population sees white & gold?

- Statistics uses data to make inferences
- Model:
  - Let p be the probability of seeing white & gold
  - Assume that individuals are independent
- Data:
  - Suppose X out of N sampled individuals see white & gold; e.g., $N = 22$, $X = 4$.
  - According to model, $X \sim Bin(N, p)$
- "Obvious" estimate of $p = X/N = 4/22 = 0.1818$
- But how is this estimate derived?

Generally useful method: maximum likelihood

- Likelihood function: probability of observed data as function of p

$$L(p) = P(X = 4) = \binom{22}{4} p^4 (1-p)^{(22-4)}, p \in (0, 1)$$

- Intuition: p that gives higher $L(p)$ is more "likely" to be correct
- Maximum likelihood estimate $\hat{p} = \arg\max L(p)$
- By differentiating

$$\log L(p) = c + 4 \log p + 18 \log(1-p)$$

  we get

$$\frac{d}{dp} \log L(p) = \frac{4}{p} - \frac{18}{1-p} = 0 \implies 4(1-p) - 18p = 0 \implies p = \frac{4}{22}$$

How could we do this numerically?

- Pretend for the moment that we did not know how to do this.
- How could we arrive at the same solution numerically?
- Basic idea: Compute $L(p)$ for various values of p and find minimum.
- To do this in R, remember that R works like a calculator:
  - The user types in an expression, R calculates the answer
  - The expression can involve numbers, variables, and functions
- For example:

```
N = 22
x = 4
p = 0.5
choose(N, x) * p^x * (1-p)^(N-x)
```

```
[1] 0.001744032
```

"Vectorized" computations

- One important distinguishing feature of R is that it operates on "vectors"

```
pvec = seq(0, 1, by = 0.01)
pvec
```

```
  [1] 0.00 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13 0.14 0.15 0.16 0.17 0.18 0.1
 [24] 0.23 0.24 0.25 0.26 0.27 0.28 0.29 0.30 0.31 0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39 0.40 0.41 0.4
 [47] 0.46 0.47 0.48 0.49 0.50 0.51 0.52 0.53 0.54 0.55 0.56 0.57 0.58 0.59 0.60 0.61 0.62 0.63 0.64 0.6
 [70] 0.69 0.70 0.71 0.72 0.73 0.74 0.75 0.76 0.77 0.78 0.79 0.80 0.81 0.82 0.83 0.84 0.85 0.86 0.87 0.8
 [93] 0.92 0.93 0.94 0.95 0.96 0.97 0.98 0.99 1.00
```

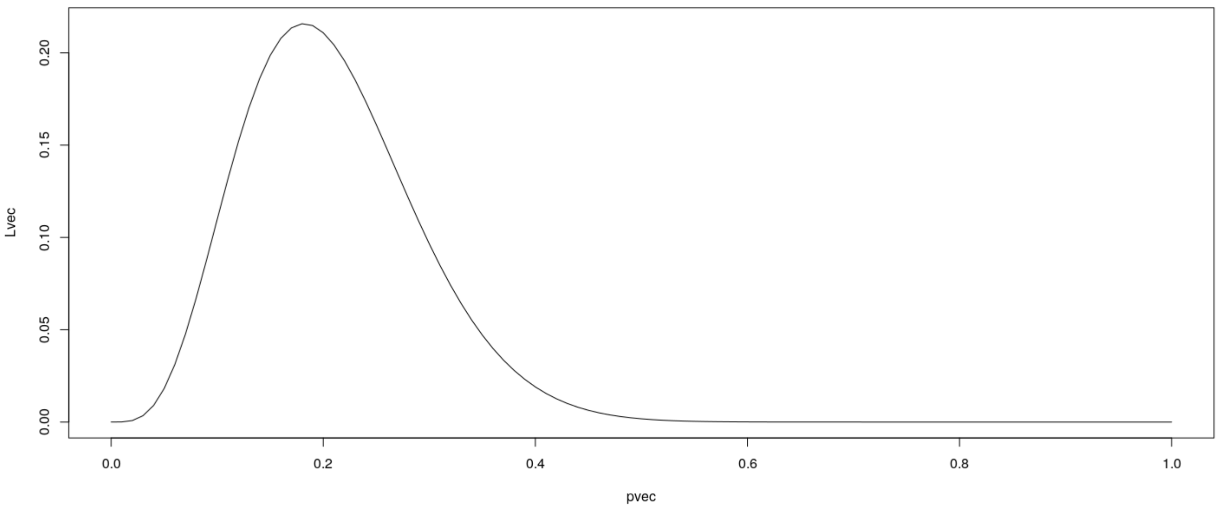
```
Lvec = choose(N, x) * pvec^x * (1-pvec)^(N-x)
Lvec
```

```
  [1] 0.000000e+00 6.104468e-05 8.135864e-04 3.424448e-03 8.981244e-03 1.816014e-02 3.112581e-02 4.7566
  [9] 6.679673e-02 8.788797e-02 1.097942e-01 1.314635e-01 1.519244e-01 1.703469e-01 1.860795e-01 1.9866
 [17] 2.078363e-01 2.135088e-01 2.157514e-01 2.147628e-01 2.108405e-01 2.043521e-01 1.957077e-01 1.8533
 [25] 1.736617e-01 1.610932e-01 1.480058e-01 1.347351e-01 1.215714e-01 1.087568e-01 9.648595e-02 8.4907
 [33] 7.412670e-02 6.421205e-02 5.519768e-02 4.708981e-02 3.987162e-02 3.350820e-02 2.795111e-02 2.3142
 [41] 1.901852e-02 1.551265e-02 1.255785e-02 1.008871e-02 8.042842e-03 6.361987e-03 4.992675e-03 3.8866
```

```
[49] 3.000817e-03 2.297533e-03 1.744032e-03 1.312274e-03 9.785206e-04 7.229017e-04 5.289691e-04 3.83257
[57] 2.748619e-04 1.950500e-04 1.369032e-04 9.500179e-05 6.514771e-05 4.412621e-05 2.950433e-05 1.94629
[65] 1.265841e-05 8.111339e-06 5.116943e-06 3.175152e-06 1.936197e-06 1.159101e-06 6.804388e-07 3.91211
[73] 2.199829e-07 1.207963e-07 6.466384e-08 3.368058e-08 1.703221e-08 8.342155e-09 3.946399e-09 1.79750
[81] 7.854421e-10 3.278832e-10 1.301292e-10 4.882107e-11 1.719852e-11 5.643294e-12 1.708098e-12 4.71272
[89] 1.167905e-13 2.551777e-14 4.799371e-15 7.529135e-16 9.440292e-17 8.910680e-18 5.800271e-19 2.27284
[97] 4.269521e-22 2.508903e-24 1.768718e-27 7.026760e-33 0.000000e+00
```

Plotting is very easy

```
plot(x = pvec, y = Lvec, type = "l")
```



Functions

- Functions can be used to encapsulate repetitive computations
- Like mathematical functions, they take arguments as input and “returns” an output

```
L = function(p) choose(N, x) * p^x * (1-p)^(N-x)
L(0.5)
```

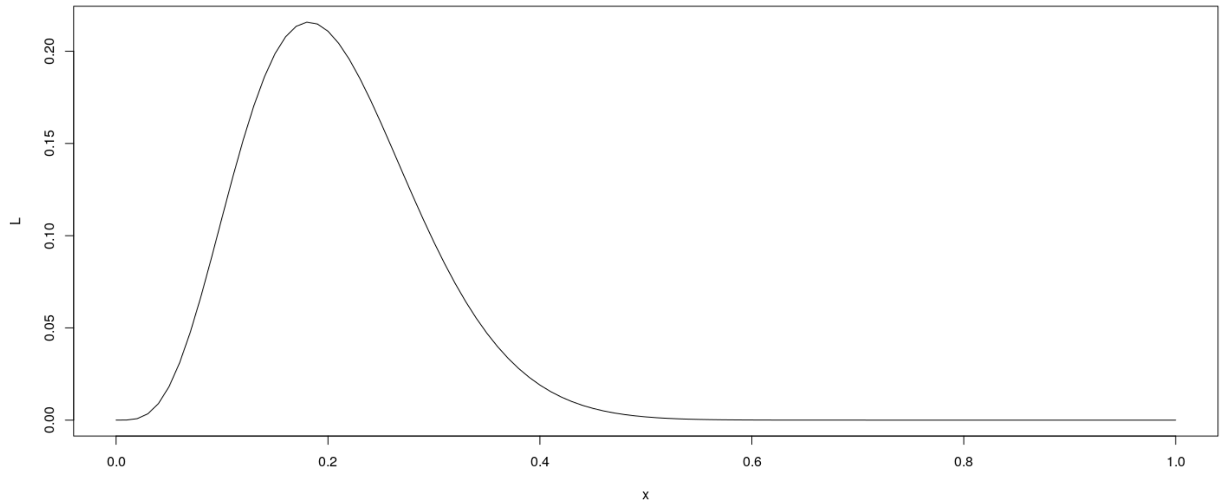
```
[1] 0.001744032
```

```
L(x/N)
```

```
[1] 0.2158045
```

Functions can be plotted directly

```
plot(L, from = 0, to = 1)
```



...and they can be numerically "optimized"

```
optimize(L, interval = c(0, 1), maximum = TRUE)
```

```
$maximum
[1] 0.1818189

$objective
[1] 0.2158045
```

A more complicated example

- Suppose $X_1, X_2, ..., X_n \sim Bin(N, p)$, and are independent
- Instead of observing each X_i, we only get to know $M = \max(X_1, X_2, ..., X_n)$
- What is the maximum likelihood estimate of p? (N and n are known, $M = m$ is observed)

To compute likelihood, we need p.m.f. of M :

$$P(M \leq m) = P(X_1 \leq m, ..., X_n \leq m) = \left[\sum_{x=0}^{m} \binom{N}{x} p^x (1-p)^{(N-x)}\right]^n$$

and

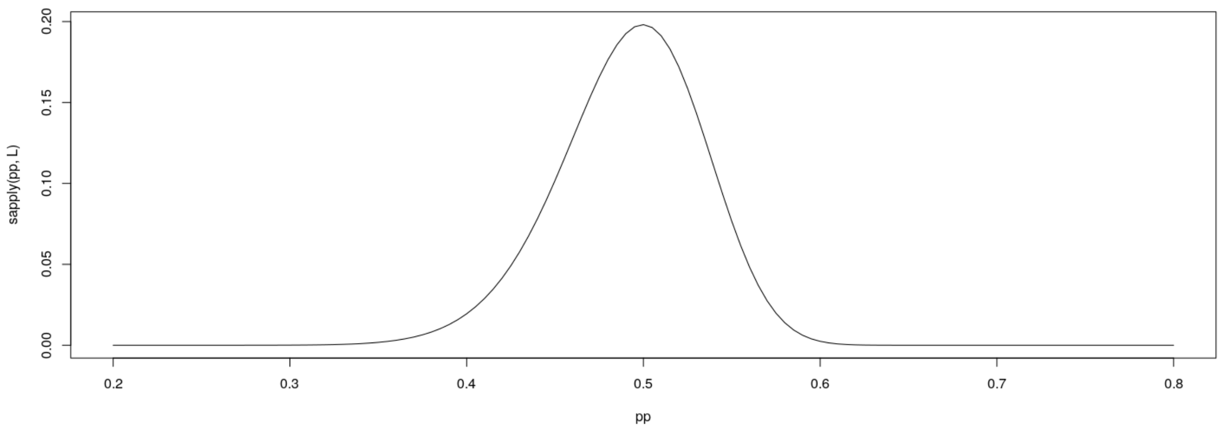
$$P(M = m) = P(M \leq m) - P(M \leq m - 1)$$

In R,

```
n = 10
N = 50
M = 30
F <- function(p, m)
{
    x = seq(0, m)
    (sum(choose(N, x) * p^x * (1-p)^(N-x)))^n
}
```

```
L = function(p)
{
    F(p, M) - F(p, M-1)
}
```

Maximum Likelihood estimate




```
optimize(L, interval = c(0, 1), maximum = TRUE)
```

```
$maximum
[1] 0.4996703

$objective
[1] 0.1981222
```

Another example: Linear regression

```
data(Davis, package = "carData")        # Davis height and weight data
str(Davis)                              # Summarize structure of data
```

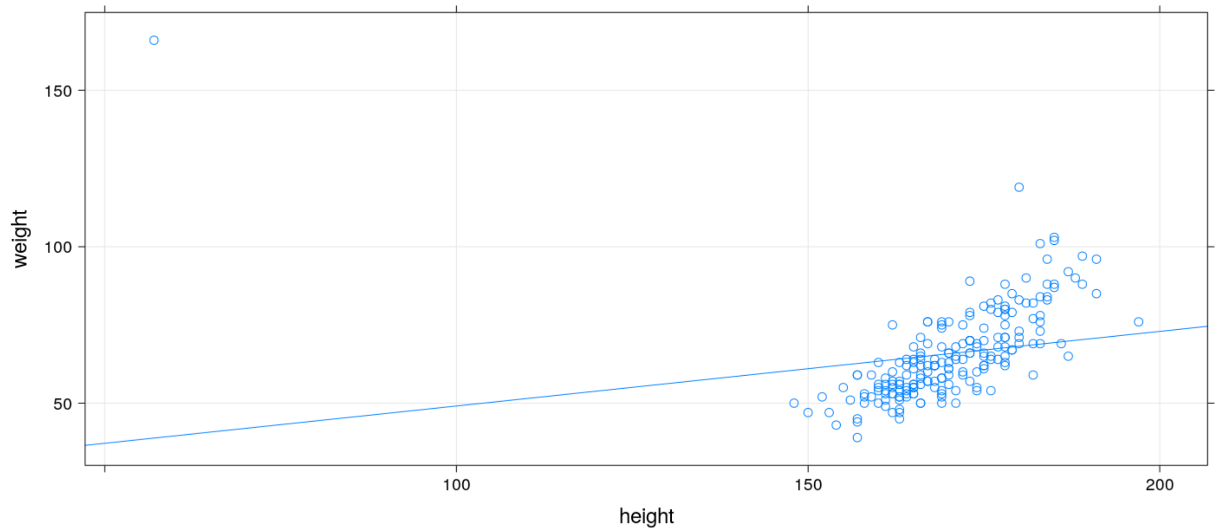
```
'data.frame':   200 obs. of  5 variables:
 $ sex   : Factor w/ 2 levels "F","M": 2 1 1 2 1 2 2 2 2 2 ...
 $ weight: int  77 58 53 68 59 76 76 69 71 65 ...
 $ height: int  182 161 161 177 157 170 167 186 178 171 ...
 $ repwt : int  77 51 54 70 59 76 77 73 71 64 ...
 $ repht : int  180 159 158 175 155 165 165 180 175 170 ...
```

```
fm <- lm(weight ~ height, data = Davis) # Regress weight on height
fm
```

```

Call:
lm(formula = weight ~ height, data = Davis)

Coefficients:
(Intercept)       height
    25.2662       0.2384
```

You should always plot the data first!

```
library(lattice)
xyplot(weight ~ height, data = Davis, grid = TRUE, type = c("p", "r"))
```



The regression model is fit by minimizing least squares

```
coef(fm) # estimated regression coefficients
```

```
(Intercept)      height 
 25.2662278   0.2384059 
```

We can confirm using a general optimizer:

```
SSE = function(beta)
{
    with(Davis,
         sum((weight - beta[1] - beta[2] * height)^2))
}
optim(c(0, 0), fn = SSE)
```

```
$par
[1] 25.3053648  0.2381936

$value
[1] 43713.12

$counts
function gradient 
      99       NA 

$convergence
[1] 0

$message
NULL
```

Fitting the regression model

`lm()` gives exact solution and more statistically relevant details

```
summary(fm)
```

```
Call:
lm(formula = weight ~ height, data = Davis)

Residuals:
    Min      1Q  Median      3Q     Max 
-23.696  -9.506  -2.818   6.372 127.145 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)   
(Intercept) 25.26623   14.95042   1.690  0.09260 . 
height       0.23841    0.08772   2.718  0.00715 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 14.86 on 198 degrees of freedom
Multiple R-squared:  0.03597,	Adjusted R-squared:  0.0311 
F-statistic: 7.387 on 1 and 198 DF,  p-value: 0.007152
```

Changing the model-fitting criteria

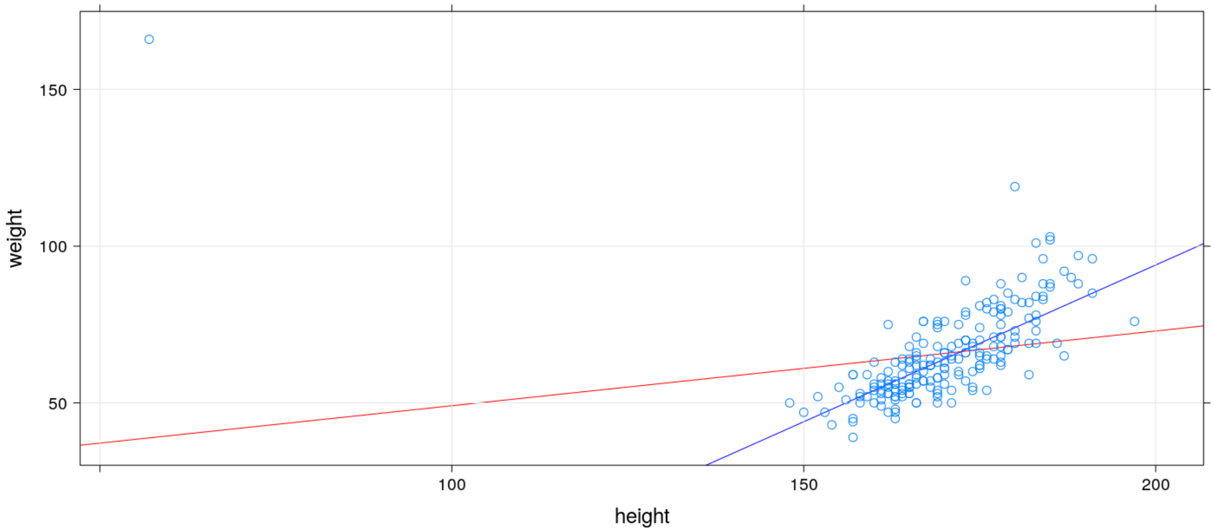
- Suppose we wanted to minimize *sum of absolute errors* instead of sum of squares
- No closed form solution any more, but general optimizer will still work:

```
SAE = function(beta)
{
    with(Davis,
         sum(abs(weight - beta[1] - beta[2] * height)))
}
opt = optim(c(0, 0), fn = SAE)
opt
```

```
$par
[1] -106.000787    1.000005

$value
[1] 1504

$counts
function gradient 
     169       NA 

$convergence
[1] 0

$message
NULL
```

This is an example of *robust regression*

```
xyplot(weight ~ height, data = Davis, grid = TRUE,
       panel = function(x, y, ...) {
           panel.abline(fm, col = "red") # squared errors
           panel.abline(opt$par, col = "blue") # absolute errors
           panel.xyplot(x, y, ...)
       })
```



R gives access to an extensive toolset

- Most standard data analysis methods are already implemented
- Can be extended by writing add-on packages
- Thousands of add-on packages are available

Drawbacks

- Learning R needs some effort
- Not point-and-click software
- Command-line interface

Good practices

- Use a good interface (R Studio is the most popular one)
- Save your code in a script file (makes it easier to reproduce later)