

## Lecture 9 - Lists, Bootstrap and Jackknife

Till now we learned about vectors, matrices, and dataframes, and we also learned about classes. Let us look a bit deeper into R.

### Lists

```
> a=c(x=1,y=5)
> a
  x y
  1 5
> a=c(1:4,10:14)
> a
 [1]  1  2  3  4 10 11 12 13 14
> a=c(x=1:4,y=10:14)
> a
  x1 x2 x3 x4 y1 y2 y3 y4 y5
  1  2  3  4 10 11 12 13 14
> a=list(x=1:4,y=10:14)
> a
 $x
 [1] 1 2 3 4

 $y
 [1] 10 11 12 13 14
> a$x
 [1] 1 2 3 4
> a$y
 [1] 10 11 12 13 14
> a[["x"]]
 [1] 1 2 3 4
> a[["y"]]
 [1] 10 11 12 13 14
> a=list(x=1:4,y=c("a","b"),z=1:10)
> a
 $x
 [1] 1 2 3 4

 $y
 [1] "a" "b"

 $z
 [1]  1  2  3  4  5  6  7  8  9 10
> a[1]
 $x
 [1] 1 2 3 4
> a[1:2]
 $x
 [1] 1 2 3 4
```

```

    $y
    [1] "a" "b"
> a[[1]]
    [1] 1 2 3 4
>

```

Using [] will give us a sublist of the list, which is still a list. Using [[]] will give us elements of the list.

We already learned about `sapply`, there is also a function called `lapply`.

```

> lapply(1:10,function(x) x^2)
[[1]]
 [1] 1

[[2]]
 [1] 4

[[3]]
 [1] 9

[[4]]
 [1] 16

[[5]]
 [1] 25

[[6]]
 [1] 36

[[7]]
 [1] 49

[[8]]
 [1] 64

[[9]]
 [1] 81

[[10]]
 [1] 100
> a=lapply(1:10,function(x) x^2)
> a[1:3]
[[1]]
 [1] 1

[[2]]
 [1] 4

[[3]]
 [1] 9
> a[[3]]
 [1] 9
> a[[1:3]]

```

```
Error: recursive indexing failed at level 2
```

```
>
```

sapply and lapply are very similar, except that sapply will turn its output into a matrix or vector in the end.

```
> a=lapply(31:40,c);a
```

```
[[1]]  
[1] 31
```

```
[[2]]  
[1] 32
```

```
[[3]]  
[1] 33
```

```
[[4]]  
[1] 34
```

```
[[5]]  
[1] 35
```

```
[[6]]  
[1] 36
```

```
[[7]]  
[1] 37
```

```
[[8]]  
[1] 38
```

```
[[9]]  
[1] 39
```

```
[[10]]  
[1] 40
```

```
> names(a)=c("a","b","c","d","e","f","g","h","i","j")
```

```
> a
```

```
$a  
[1] 31
```

```
$b  
[1] 32
```

```
$c  
[1] 33
```

```
$d  
[1] 34
```

```
$e  
[1] 35
```

```
$f  
[1] 36
```

```

$g
[1] 37

$h
[1] 38

$i
[1] 39

$j
[1] 40
> a$d
[1] 34
> a["e"]
 $e
[1] 35
> b=lapply(a,function(x) x-20)
> b
 $a
[1] 11

 $b
[1] 12

 $c
[1] 13

 $d
[1] 14

 $e
[1] 15

 $f
[1] 16

 $g
[1] 17

 $h
[1] 18

 $i
[1] 19

 $j
[1] 20
> sapply(a,function(x) x-20)
  a  b  c  d  e  f  g  h  i  j
11 12 13 14 15 16 17 18 19 20
>

```

How to turn a list into a vector?

```
> b
```

```

$a
[1] 11

$b
[1] 12

$c
[1] 13

$d
[1] 14

$e
[1] 15

$f
[1] 16

$g
[1] 17

$h
[1] 18

$i
[1] 19

$j
[1] 20
> unlist(b)
  a  b  c  d  e  f  g  h  i  j
11 12 13 14 15 16 17 18 19 20
>
Another extremely usefull function is tapply
> setwd("~/R-course-2006/lecture8")
> list.files()
[1] "factorial.txt"          "oneway.txt"           "Rcourse8_no_output.tm"
[4] "Rcourse8_no_output.tm~" "Rcourse8.pdf"         "Rcourse8.tm"
[7] "Rcourse8.tm~"           "regression.txt"      "twoway.txt"
> fact=read.table("factorial.txt",sep="\t",head=T)
> fact
  growth diet  coat
1     6.6   A light
2     7.2   A light
3     6.9   B light
4     8.3   B light
5     7.9   C light
6     9.2   C light
7     8.3   A  dark
8     8.7   A  dark
9     8.1   B  dark
10    8.5   B  dark
11    9.1   C  dark

```

```

12  9.0  C  dark
> tapply(fact$growth,fact$diet,c)
$A
[1] 6.6 7.2 8.3 8.7

$B
[1] 6.9 8.3 8.1 8.5

$C
[1] 7.9 9.2 9.1 9.0
> tapply(fact$growth,fact$coat,c)
$dark
[1] 8.3 8.7 8.1 8.5 9.1 9.0

$light
[1] 6.6 7.2 6.9 8.3 7.9 9.2
> tapply(fact$growth,fact$diet,median)
  A    B    C
7.75 8.20 9.05
> tapply(fact$growth,fact$coat,c)
$dark
[1] 8.3 8.7 8.1 8.5 9.1 9.0

$light
[1] 6.6 7.2 6.9 8.3 7.9 9.2
> tapply(fact$growth,fact$coat,median,simplify=F)
$dark
[1] 8.6

$light
[1] 7.55
> tapply(fact$growth,fact$coat,median,simplify=T)
  dark light
 8.60  7.55
>

```

dataframe are also lists, but special lists.

```

> fact
  growth diet  coat
1     6.6   A light
2     7.2   A light
3     6.9   B light
4     8.3   B light
5     7.9   C light
6     9.2   C light
7     8.3   A  dark
8     8.7   A  dark
9     8.1   B  dark
10    8.5   B  dark
11    9.1   C  dark
12    9.0   C  dark
> fact[["growth"]]

```

```

      [1] 6.6 7.2 6.9 8.3 7.9 9.2 8.3 8.7 8.1 8.5 9.1 9.0
> fact[["coat"]]
      [1] light light light light light light dark  dark  dark  dark  dark  dark
Levels: dark light
> fact[1:2]
      growth diet
1         6.6   A
2         7.2   A
3         6.9   B
4         8.3   B
5         7.9   C
6         9.2   C
7         8.3   A
8         8.7   A
9         8.1   B
10        8.5   B
11        9.1   C
12        9.0   C
> lapply(fact,c)
      $growth
      [1] 6.6 7.2 6.9 8.3 7.9 9.2 8.3 8.7 8.1 8.5 9.1 9.0

      $diet
      [1] 1 1 2 2 3 3 1 1 2 2 3 3

      $coat
      [1] 2 2 2 2 2 2 1 1 1 1 1 1
> attributes(fact)
      $names
      [1] "growth" "diet"  "coat"

      $class
      [1] "data.frame"

      $row.names
      [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12"
>

```

The `attributes` of an object can cause R to treat it differently. Two attributes that we already encountered are `class`, and `names`. `fact` is a list, but because the class is “data.frame”, it is treated in a special way.

```

> a=matrix(1:12,3,4)
> a
      [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> attributes(a)
      $dim
      [1] 3 3
>

```

all that separates a vector from a matrix is the `dim` attributes.

```

> attributes(a)$dim=c()
> a
  [1]  1  2  3  4  5  6  7  8  9 10 11 12
> attributes(a)$dim=c(3,4)
> a
      [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> attributes(a)$dim=c(4,3)
> a
      [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
>

```

## Bootstrapping

The principle of the bootstrap method is that the data the we gathered can represent the external world.

In order to get a distribution of possible values that we could have measured, we re-sample from the data.

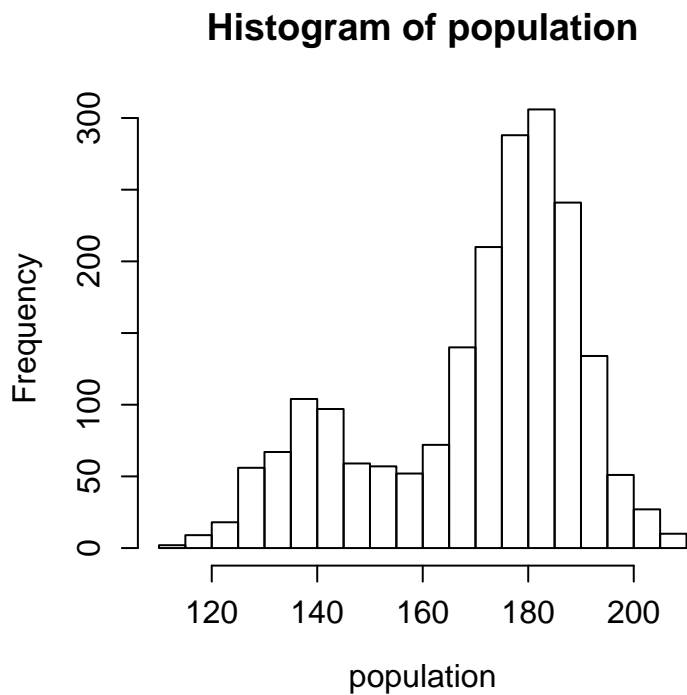
### Example: Median

Let us say that we have a population of 2000 people, whose heights are the following:

```

> population=c(rnorm(1500,mean=180,sd=10),rnorm(500,mean=140,sd=10))
> hist(population,n=30);v()

```



>

Now let us assume that we measure the heights of 20 randomly chosen people exactly:

```
> measurements=sample(population,20)
```

```
> measurements
```

```
[1] 182.6651 195.6872 182.5375 174.9788 183.1086 164.1614 127.4974 178.6637
[9] 184.6356 178.1290 179.3890 170.3763 138.7274 183.9492 182.9451 181.6166
[17] 191.0370 179.0268 124.8710 128.2074
```

>

We want to know the median of the distribution. To approximate that, we measure the median of the measurements:

```
> median(measurements)
```

```
[1] 179.2079
```

>

We know now the median of the measurements. How wrong are we?

If we had the whole population at our disposal, we could do the process of sampling many times, and see the distribution of the median:

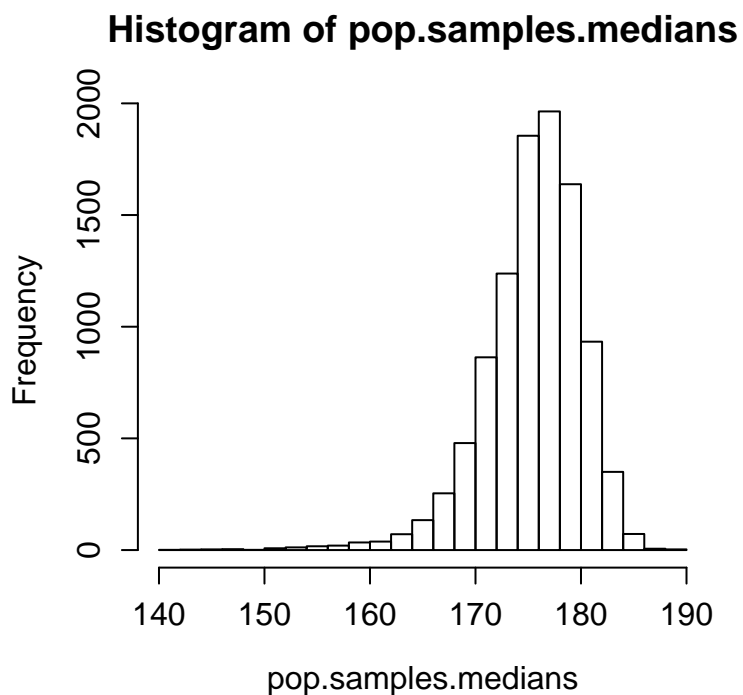
```
> pop.samples=t(sapply(1:10000,function(i) sample(population,20) ))
```

```
> dim(pop.samples)
```

```
[1] 10000    20
```

```
> pop.samples.medians=apply(pop.samples,1,median)
```

```
> hist(pop.samples.medians,n=30);v()
```



```
> median(population)
```

```
[1] 176.1048
```

```
>
```

We can see that we're likely to be about 10cm off the real median.

But we don't have the real population. We only have our samples. To estimate the distribution of medians, we sample from the sample, instead of from the real population:

```
> sample.samples=t(sapply(1:10000,function(i) sample(measurements,20,rep=T) ))
```

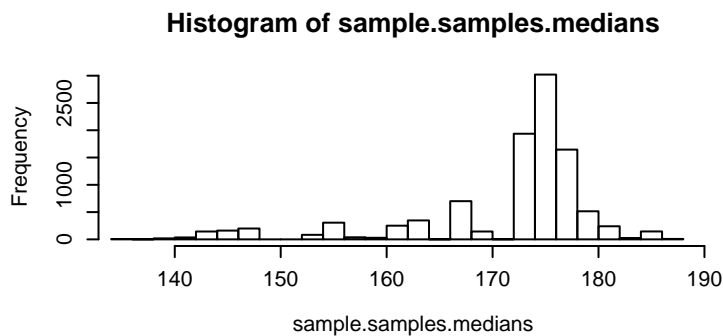
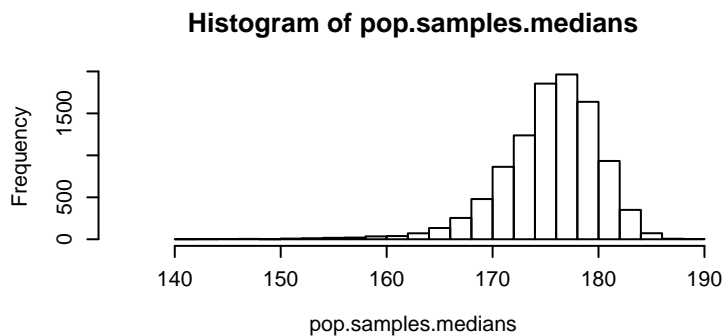
```
> sample.samples.medians=apply(sample.samples,1,median)
```

```
> layout(
```

```
matrix(1:2,2,1);par(cex=0.7);hist(pop.samples.medians,n=30,xlim=c(135,190)
```

```
);hist(sample.samples.medians,n=30,xlim=c(135,190));v();layout(1)
```

```
+
```



```
>
```

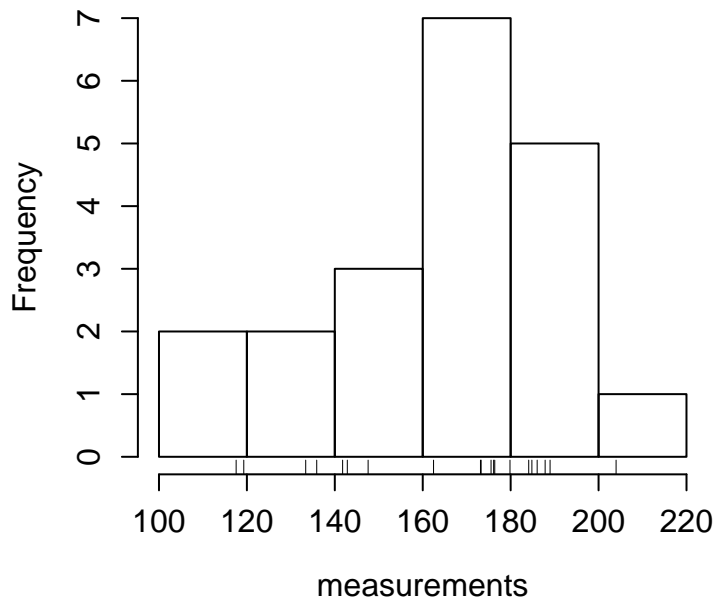
As you see, we do not get the same distribution! The first distribution was created by sampling from the real population. The second distribution was sampled from our sample.

## Parametric bootstrap

Instead of using the sample as our model for the world, we can base a model on the sample, and then sample from it. For example, let us look at our sample.

```
> hist(measurements);rug(measurements);v()
```

## Histogram of measurements



>

It obviously does not look normal - it has too much of a tail on the left.

We could still try to fit the best normal distribution we can find, and then bootstrap.

```
> a=measurements-mean(measurements)
> a=a/sd(a)
> ks.test(a,pnorm)
```

One-sample Kolmogorov-Smirnov test

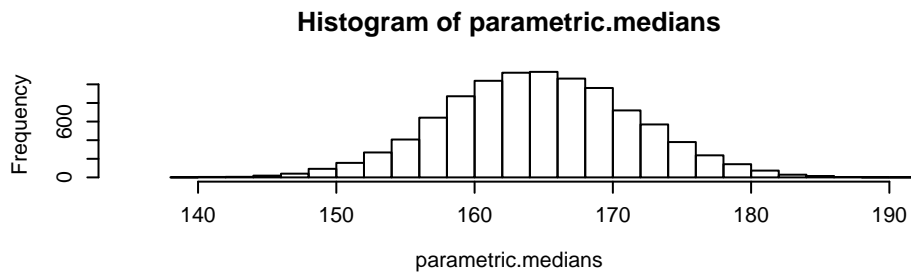
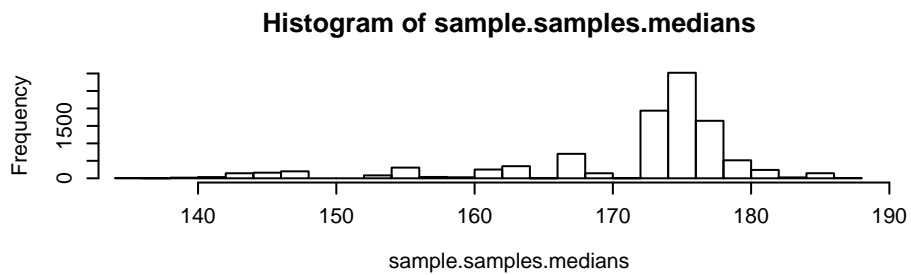
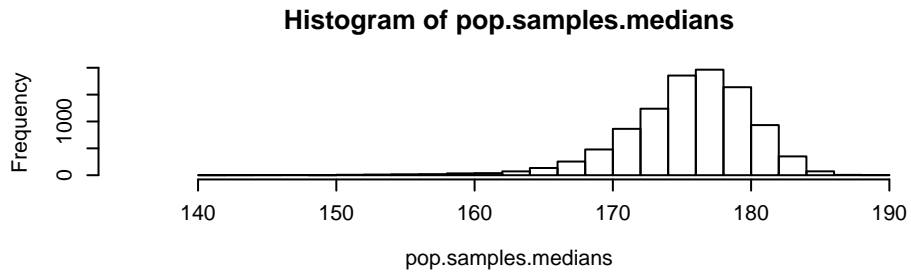
```
data: a
D = 0.2335, p-value = 0.2254
alternative hypothesis: two.sided
```

>

You can see that because of the small sample size, the KS test is actually not significant.

```
> mm=mean(measurements);mm
[1] 164.5417
> msd=sd(measurements);msd
[1] 25.19489
> parametric=matrix( rnorm(10000*20,mean=mm,sd=msd), 10000, 20)
> parametric.medians=apply(parametric,1,median)
> layout(
matrix(1:3,3,1));par(cex=0.7);hist(pop.samples.medians,n=30,xlim=c(135,190)
);hist(sample.samples.medians,n=30,xlim=c(135,190)
);hist(parametric.medians,n=30,xlim=c(135,190));v(width=5,height=5);layout(1)
```

+



```
> sd(sample.samples.medians)
```

```
[1] 8.75055
```

```
> sd(parametric.medians)
```

```
[1] 6.861419
```

```
> sd(pop.samples.medians)
```

```
[1] 4.687592
```

```
>
```

## Jackknife

The Jackknife method was invented before the bootstrap method, and is very similar.

Instead of taking random samples from the distribution, we drop each of our sample points.

```
> jack= t( sapply(1:length(measurements),function(i) measurements[-i] ) )
```

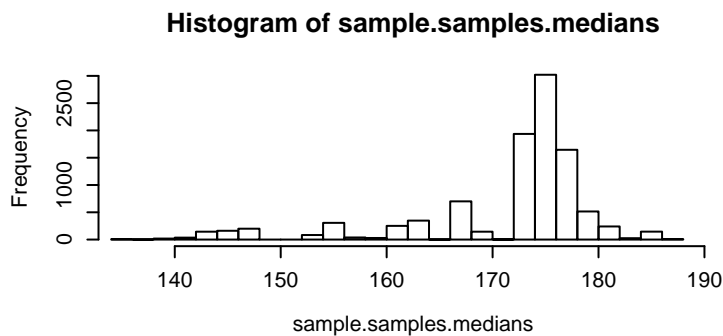
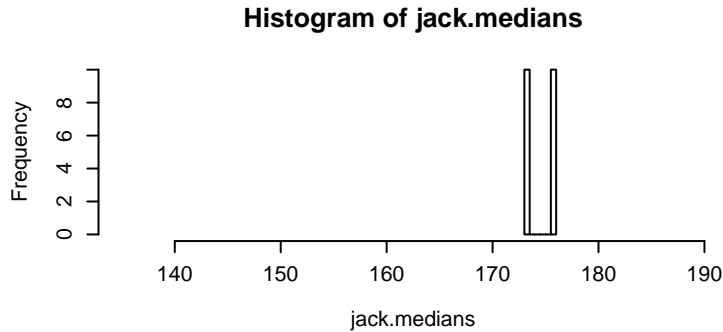
```
> jack[1:4,1:4]
```

```
      [,1]      [,2]      [,3]      [,4]
[1,] 186.0301 162.4606 176.1285 133.3787
[2,] 184.1049 162.4606 176.1285 133.3787
[3,] 184.1049 186.0301 176.1285 133.3787
[4,] 184.1049 186.0301 162.4606 133.3787
```

```

> jack.medians=apply( jack, 1, median )
> layout( matrix(1:2,2,1));par(cex=0.7);hist(jack.medians,xlim=c(135,190)
) ;hist(sample.samples.medians,n=30,xlim=c(135,190));v();layout(1)
+

```



>

We can see that the jackknife estimate of the error is too small. In fact it is too small by a factor of approximately  $\sqrt{20}$ , or  $\sqrt{(n-1)^2/n}$ .

```

> sd(jack.medians)
[1] 1.158305
> sd( jack.medians)*sqrt(20)
[1] 5.180097

```

It is actually clear that the median will move between 1 2 or 3 points at most:

```

> median( 1:3)
[1] 2
> median(1:2)
[1] 1.5
> median(c(1,3))
[1] 2
> median( 2:3)
[1] 2.5
>

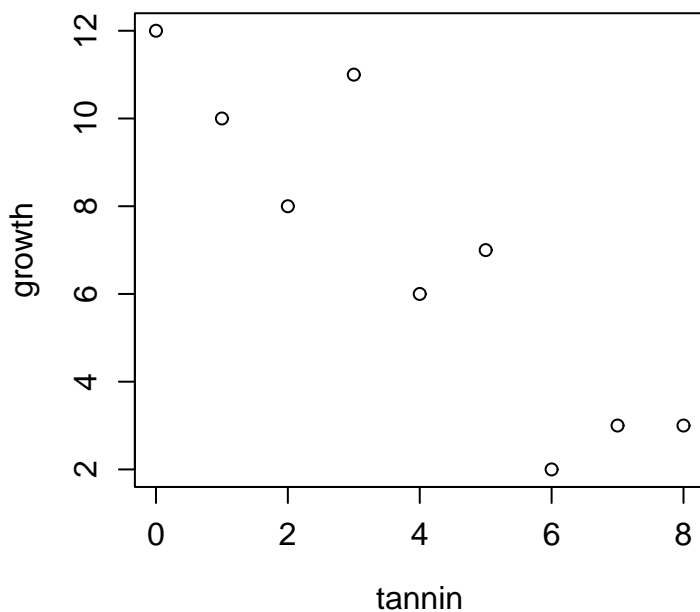
```

And as we see in the above example, we get just 2 points.

```
> table( jack.medians )
jack.medians
173.280953077412 175.538905170459
          10          10
>
```

Because the median has this property that it “jumps”, the jackknife is a bad choice in this case. Let us take another example:

```
> a=read.table("/home/dirk/data/regression.txt",head=T)
> plot(growth~tannin,data=a);v()
```



```
> reg=lm(growth~tannin,data=a)
> reg

Call:
lm(formula = growth ~ tannin, data = a)

Coefficients:
(Intercept)      tannin
    11.756         -1.217

> names(reg)
 [1] "coefficients" "residuals"    "effects"      "rank"
 [5] "fitted.values" "assign"       "qr"           "df.residual"
 [9] "xlevels"      "call"        "terms"       "model"

> reg$coefficients[2]
```

```

    tannin
-1.216667
>

```

Let us say that we would like to know how exact this estimate of the slope is

First, let us write a function that given growth and tannin tells us the slope:

```

> slope=function(gr,ta){reg=lm(gr~ta);reg$coeff[2]}
> slope(a$gro,a$tan)
    ta
-1.216667
>

```

Now let us write a function that given indexes, calculates the slope for those points of data from a:

```

> index.slope=function(i) slope( a$gr[i], a$tan[i] )
> a

```

	growth	tannin
1	12	0
2	10	1
3	8	2
4	11	3
5	6	4
6	7	5
7	2	6
8	3	7
9	3	8

```

> index.slope(1:9)

```

```

    ta
-1.216667

```

```

> index.slope(1:3)

```

```

    ta
-2

```

```

> index.slope(-1)

```

```

    ta
-1.190476

```

```

>

```

Now the jackknife is very simple:

```

> jack.slopes=sapply(-(1:9), index.slope)
> jack.slopes

```

ta	ta	ta	ta	ta	ta	ta		
ta	-1.190476	-1.253133	-1.270270	-1.161359	-1.216667	-1.242038	-1.117117	-
	1.200501							
ta	-1.321429							

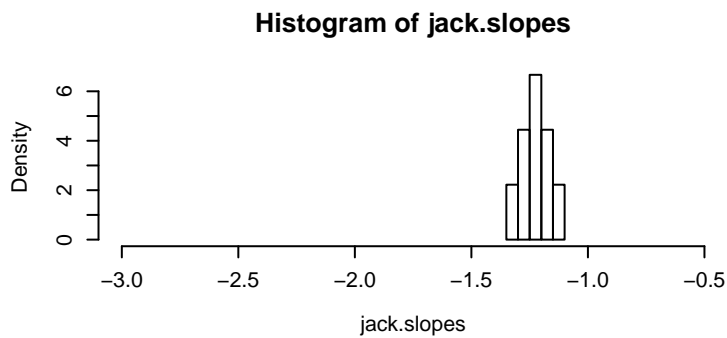
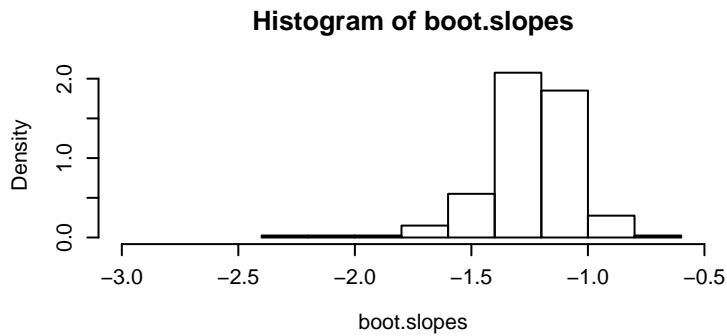
```

>

```

Bootstrap is a bit more complicated:

```
> boot=t( sapply(1:200,function(i) sample(1:9,rep=T) ) )
> boot.slopes=apply(boot,1,index.slope)
> layout(matrix(1:2,2,1));par(cex=0.7);hist(boot.slopes,prob=T,xlim=c(-3,-0.5))
> hist(jack.slopes,prob=T,xlim=c(-3,-0.5));v();layout(1)
```



```
> sd(boot.slopes)
[1] 0.1943785
> sd(jack.slopes)
[1] 0.06090926
> sd(jack.slopes)*sqrt(9)
[1] 0.1827278
>
```

## Confidence intervals

A general rule of thumb says that to estimate the error, 200 bootstrap samples are enough. To estimate 95% confidence intervals, one needs around 1000.

```
> boot=t( sapply(1:1000,function(i) sample(1:9,rep=T) ) )
> boot.slopes=apply(boot,1,index.slope)
> quantile(boot.slopes,0.025)
 2.5%
-1.609121
> quantile(boot.slopes,0.975)
97.5%
```

-0.9084386

>

There are actually more accurate methods for calculating confidence intervals using quantiles. It can be done with the function `bcanon` in the library `bootstrap`:

> `library(bootstrap)`

```
Error in library(bootstrap) : there is no package called 'bootstrap'
```

> `install.packages("bootstrap")`

```
Warning in install.packages("bootstrap") : argument 'lib' is missing: using  
/home/michael/lib/R
```

```
Warning message:
```

```
unable to resolve 'cran.r-mirror.de'.
```

```
Warning: unable to access index for repository http://cran.r-  
mirror.de/src/contrib
```

```
Warning message:
```

```
no package 'bootstrap' at the repositories in: download.packages(pkgs, destdir  
= tmpd, available = available,
```

> `bcanon(1:9,1000,index.slope)`

```
$confpoints
```

```
      alpha  bca point  
[1,] 0.025 -1.6774194  
[2,] 0.050 -1.5909091  
[3,] 0.100 -1.4782609  
[4,] 0.160 -1.4000000  
[5,] 0.840 -1.0920330  
[6,] 0.900 -1.0520833  
[7,] 0.950 -1.0061728  
[8,] 0.975 -0.9530201
```

```
$z0
```

```
[1] -0.06521854
```

```
$acc
```

```
[1] -0.001202818
```

```
$u
```

```
[1] -1.190476 -1.253133 -1.270270 -1.161359 -1.216667 -1.242038 -1.117117  
[8] -1.200501 -1.321429
```

```
$call
```

```
bcanon(x = 1:9, nboot = 1000, theta = index.slope)
```

```
Warning message:
```

```
multi-argument returns are deprecated in: return(confpoints, z0, acc, u, call  
= call)
```

>