Tutorial 6 | Simulation and Profiling

The str Function

str ('structure') - a diagnostic function that compactly displays the internal structure of an R object. It is almost like a help page but faster and shorter, and also can be applied to any type of object not just functions

```
# Let's suppose I want to know what the fucntion mean takes as
#arguments
str(mean)
```

```
## function (x, ...)
```

```
# Or maybe if I have a vector and I want to look at it briefly
x<-rnorm(100,2,4)
str(x)
```

```
##  num [1:100] -2.43 2.12 6.69 3.72 6.52 ...
```

```
# And in this case summary will produce kind of the same output:
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -4.906  -1.515   2.061   2.190   5.369  18.153
```

```
# But what if I have a really long vector of factors?
# f is a vector of factors with 40 levels and each one is repeated
# 10 times
f<- gl(40, 10)
str(f)
```

```
##  Factor w/ 40 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```r
# Now this gives us again a very compact information on what f is all
# about, and let's see what does summary will give us as an output?
summary(f)
```

```
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
## 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
## 27 28 29 30 31 32 33 34 35 36 37 38 39 40
## 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

```r
# Well this one isn't really formative tbh
```

It also gives a little more information that might be useful about data frames as well:

```r
library(datasets)
str(airquality)
```

```
## 'data.frame':    153 obs. of  6 variables:
##  $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
##  $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
##  $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
##  $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
##  $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
##  $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
```

Simulation

Generating Random Variables

Functions for generating random variables for different kind of distribution patterns:

- **rnorm**: generation of random Normal values with the given mean and standard deviation
- **dnorm**: evaluate the Normal probability density (with the given mean/standard deviation) at a point (or vector of points)
- **pnorm**: evaluate the cumulative distribution function for a Normal distribution
- **rpois**: generation of random Poisson variables with a given rate

Probability distribution functions usually have four fucntions associated with them. The functions start with:

- **d** for density
- **r** for random number generation
- **p** for cumulative distribution
- **q** for quantile function

```r
# Here we generate the vector of random normal variables, with the
# default stdev = 1 and mean = 0:
x <- rnorm(5)
x
```

```
## [1]  0.39476880 -0.39640247  1.43648236 -0.08090378 -1.92248814
```

```r
# And here is how we can modify the arguments of the function in
#order to make it random normal variables with the mean 10 and stdev 2:
x <- rnorm(5,20,2)
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   17.72   18.95   20.39   20.11   20.79   22.69
```

It is very important to set the seed prior to using random number generations, so that you result are reproducible. The importance of reproducibility of your results were discussed in the previous lesson(If you get an error or a bug you need to be able to reproduce the result you got in order to analyze the problem)

```r
# You set the seed by providing any integer values to this function
set.seed(1)
rnorm(5)
```

```
## [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

```r
# Now you see that the second time we call rnorm the numbers are different
rnorm(5)
```

```
## [1] -0.8204684  0.4874291  0.7383247  0.5757814 -0.3053884
```

```r
# However, if I reset the seed to be 1 again I will get the same
# rnorm variables as i got the first time:
set.seed(1)
rnorm(5)
```

```
## [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

As it was said before there are of course other distributions that can be generated besides Normal distribution. For example, let's take Poisson distribution:

```r
x<-rpois(15,1)
x
```

```
##  [1] 0 0 1 1 2 1 1 4 1 2 3 0 1 0 0
```

```r
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   1.000   1.133   1.500   4.000
```

Generating Random Numbers from a Linear Model

Suppose we want to simulate from the following linear model:

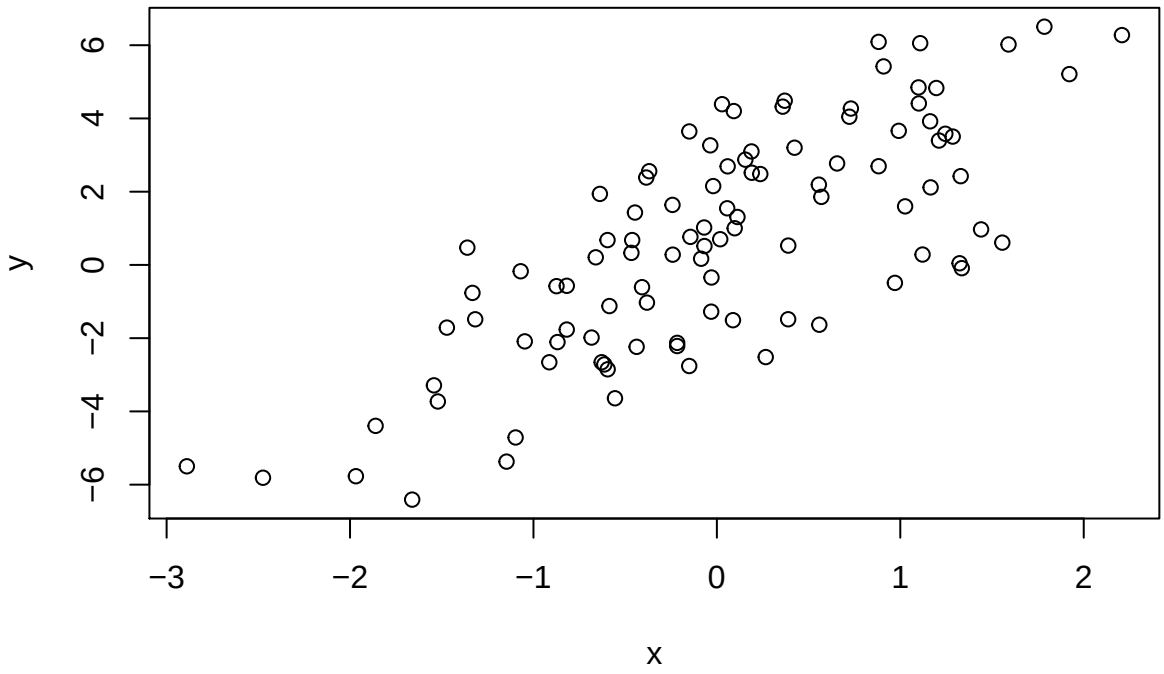
$$y = \beta_0 + \beta_1 x + \varepsilon$$

where $\varepsilon \sim N(0, 2^2)$ (normal distribution with stdev 2). Assume $x \sim N(0, 1^2), \beta_0 = 0.5, \beta_1 = 2$

```r
set.seed(20)
x<-rnorm(100)
e<-rnorm(100,0,2)
y<-0.5+2*x+e
summary(y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -6.4084 -1.5402  0.6789  0.6893  2.9303  6.5052
```

```r
plot(x,y)
```

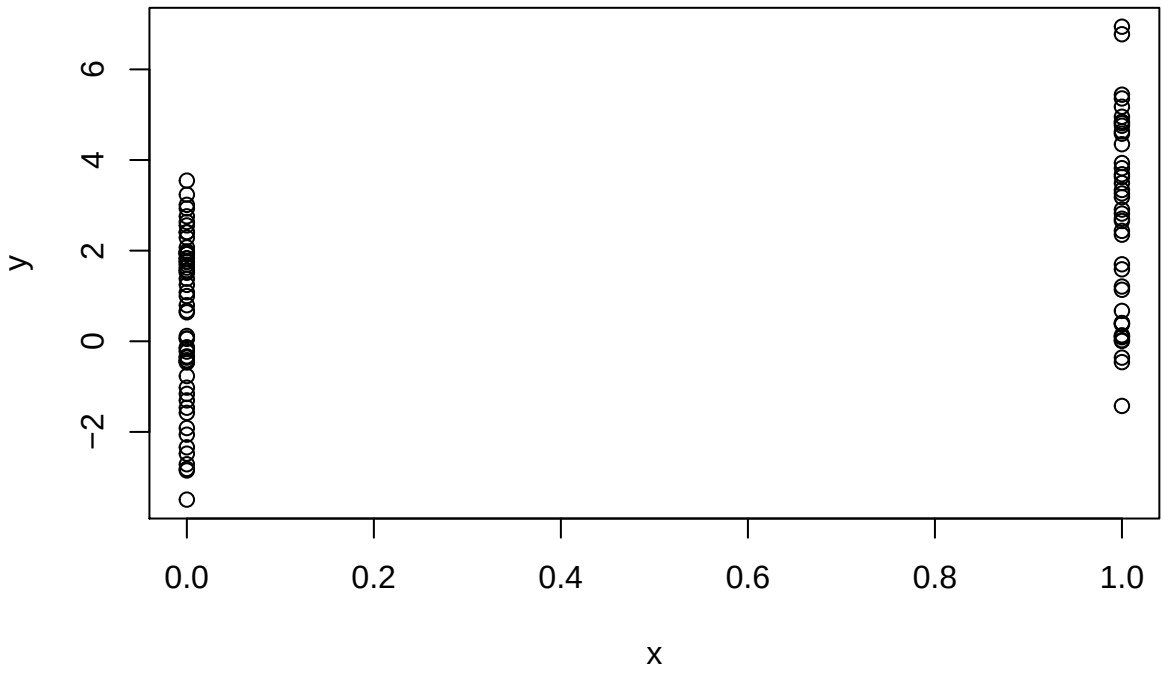


You can also use the same approach for other types of distributions. For example, let's assume you need a binomial distribution (ex. genders).

```
set.seed(10)
x<-rbinom(100,1,0.5)
e<-rnorm(100,0,2)
y<-0.5+2*x+e
summary(y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -3.4936 -0.1409  1.5767  1.4322  2.8397  6.9410
```

```
plot(x,y)
```



Now let's take a mre complicated model as an example. Let's take Poisson distribution where:

$$Y \sim Poisson(\mu)$$

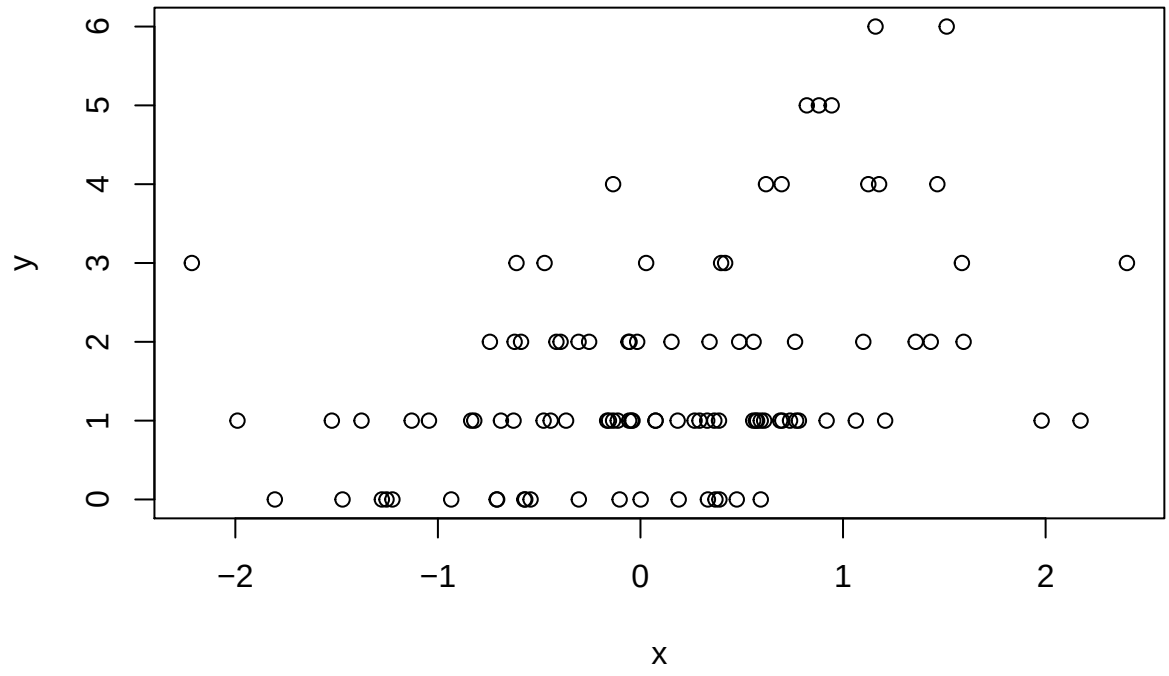
$$log(\mu) = \beta_0 + \beta_1 * x$$

$$\beta_0 = 0.5, \beta_1 = 0.3$$

```
set.seed(1)
x<-rnorm(100)
log.mu<-0.5+0.3*x
y<-rpois(100,exp(log.mu))
summary(y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    1.00    1.00    1.55    2.00    6.00
```

```
plot(x,y)
```



Random Sampling

The **sample** function draws randomly from a specified set of scalar objects allowing you to sample from arbitrary distributions. For example:

```r
set.seed(1)
# Let's suppose we need 4 random integers from the range of 1 to 10
sample(1:10,4)
```

```
## [1] 9 4 7 1
```

```r
# Of course it could be not only numbers
sample(letters, 5)
```

```
## [1] "b" "w" "k" "n" "r"
```

```r
# If you wont specify the number of elements to sample, you would
# get a permutation of numbers in a range
sample(1:10)
```

```
##  [1]  3  1  5  8  2  6 10  9  4  7
```

```r
# You can also make samples with repetitions(replacement)
sample(1:10,replace = TRUE)
```

```
##  [1]  5  9  9  5  5  2 10  9  1  4
```

R Profiler

Profiling is a systematic way to examine how much time is spend in different parts of a program. It is useful when trying to optimize the code, because for example even if your current code runs fine, when you have to put it into a larger system or have to iterate through it, let's say 5000 times, will this be fast enough? However, what you need to always remember is that optimizing your code should really be the LAST thing you do!

Using system.time()

system.time() - takes an arbitrary expression and returns the amount of time taken to evaluate the expression.

- If there is an error in the code, it will return the time until the error occured
- **user time**: time charged to the CPU(s) for this expression
- **elapsed time**: "wall clock" time, or in other words the time that you experience

Usually, the user time and elapsed time are relatively the same, for straight computing tasks.

- Elapsed time is greater if the CPU spends a lot of time waiting around (for example for other backfround tasks to complete)
- Elapsed time is smaller if you machine uses multiple cores/processors. It usually happens when Multi-threaded BLAS(Basic Linear Algebra Stadard libraries) libraries are used or parallel processing via the parallel package

```r
# Elapsed time > user time (because in this case it also depends on
# the internet connection etc)
system.time(readLines("http://www.jhsph.edu"))
```

```
##    user  system elapsed
##    0.11    0.02    1.77
```

You can also put the larger pieces of code into system.time() function by providing it in curly braces:

```r
system.time({
  r<-numeric(1000)
  for (i in 1:1000){
    a<-rnorm(1000)
    r[i]<-mean(x)
  }
})
```

```
##    user  system elapsed
##    0.06    0.00    0.06
```

Note however, that system.time() function assumes that you already know where the problem is and can call the function on it.
But what if you don't know?

Using Rprof()

The Rprof() function starts the R profiler

- R must be compiled with profiler support(but usually this is already the case)
- The summaryRprof() function summarizes the output from Rprof(otherwise it is not readable)
  DO NOT use system.time() and Rprof() together or you'll be sad

Since the output of the Rprof() function isn't quite readable we need to use summaryRprof():

- It tabulates the R profiler output and calculates how much time is spend in which function
- There are two methods for normalizing the data:
- "by.total": divides the tome spend in each function by the total run time
- "by.self": does the same but first subtracts out time spent in functions above in the call stack