Week 4

`str` **function**:

- Compactly displays the internal structure of an R object.
- A diagnostic function and an alternative to `summary`.
- It is especially well suited to compactly display the (abbreviated) contents of (possibly nested) lists.
- Roughly one line per basic object.

Generating random numbers:

Functions for probability distributions in R:

- `rnorm`: generate random normal variates with a given mean and std. deviation.
- `dnorm`: evaluate the normal probability density (with given mean and std. deviation) at a point (or vector of points).
- `pnorm`: evaluate the cumulative distribution function for a normal distribution.
- `rpois`: generate random Poisson with a given rate.

Probability distribution functions usually have four functions associated with them. The function are prefixed with a:

- `d` for density;
- `r` for random number generation.
- `p` for cumulative distribution.
- `q` for quantile function.

Working with the normal distribution requires using four functions:

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(x, mean = 0, sd = 1)
```

If φ is the cumulative distribution function for a std. normal distribution, then `pnorm(q)` = $\varphi(q)$ and `qnorm(p)` = $\varphi^{-1}(p)$.

Setting the random number seed with `set.seed` ensures reproducibility.

```
set.seed(1)
rnorm(5)
# [1] -0.6264538  0.1836433 -0.8356286  1.5952808
# [5]  0.3295078
rnorm(5)
# [1] -0.8204684  0.4874291  0.7383247  0.5757814
# [5] -0.3053884
set.seed(1)
rnorm(5)
# [1] -0.6264538  0.1836433 -0.8356286  1.5952808
# [5]  0.3295078
```

Always set the random number seed when conducting a simulation!

Generating Poisson data:

```
rpois(10, 1)
# [1] 3 1 0 1 0 0 1 0 1 1
rpois(10, 2)
# [1] 6 2 2 1 3 2 2 1 1 2
rpois(10, 20)
# [1] 20 11 21 20 20 21 17 15 24 20

ppois(2, 2)       # Cumulative distribution
# [1] 0.6766764  # Pr(x <= 2)
ppois(4, 2)
# [1] 0.947347   # Pr(x <= 4)
ppois(6, 2)
# [1] 0.9954662  # Pr(x <= 6)
```

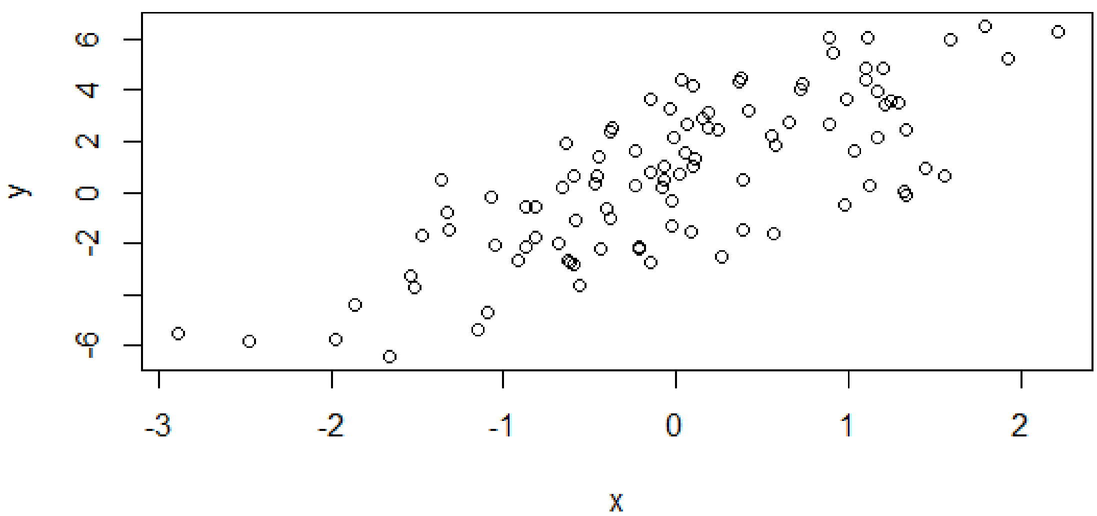
Generating random numbers from a linear model

Suppose we want to simulate from the following linear model:

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

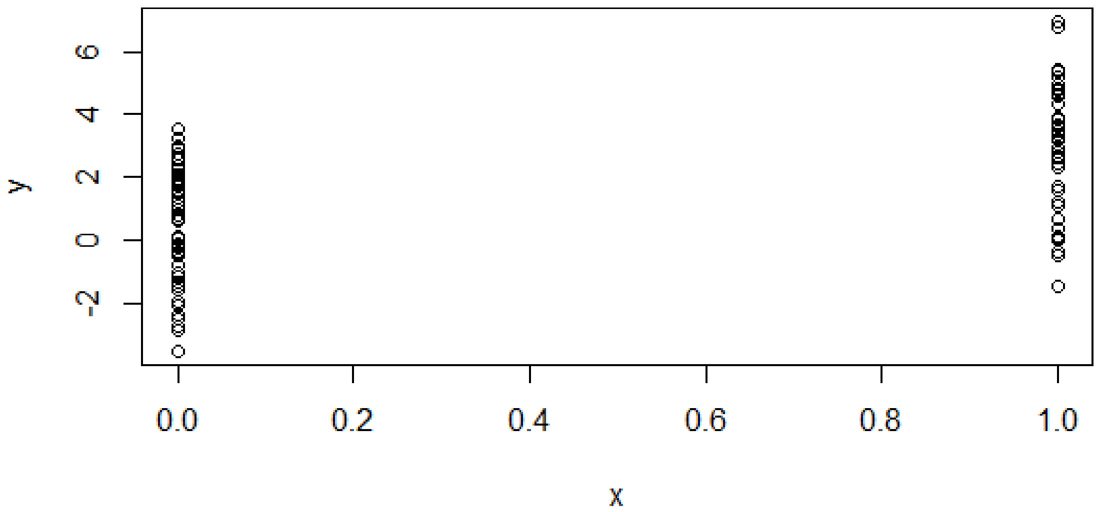
where $\varepsilon \sim \mathcal{N}(0, 2^2)$. Assume $x \sim \mathcal{N}(0, 1^2)$, $\beta_0 = 0.5$ and $\beta_1 = 2$.

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



What if `x` is binary?

```r
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)
```



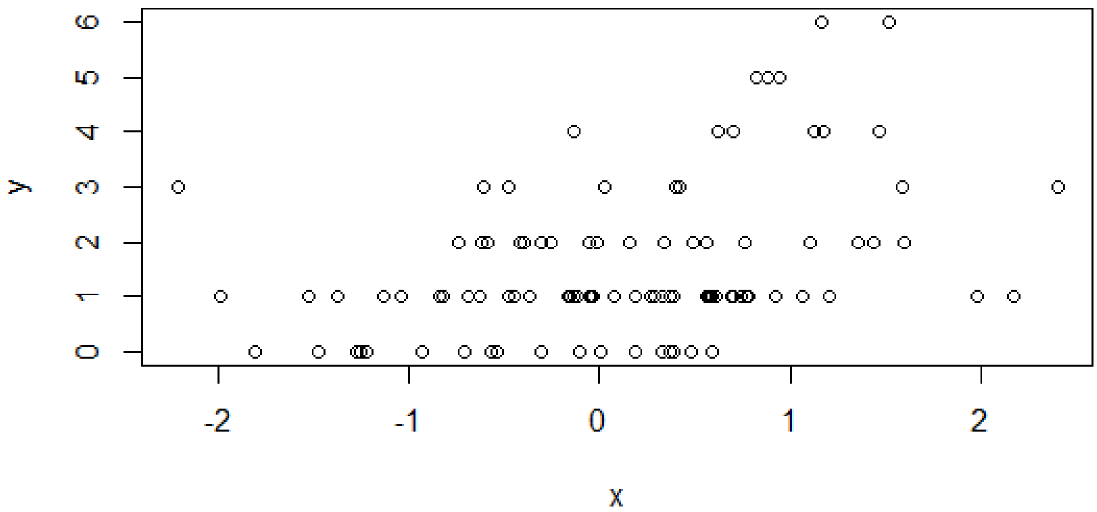
Generating random numbers from a Generalized Linear Model

Suppose we want to simulate from a Poisson model where:

$$Y \sim \text{Poisson}(\mu), \;\; \log \mu = \beta_0 + \beta_1 x$$

where $\beta_0 = 0.5$ and $\beta_1 = 0.3$.

```r
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.

```
set.seed(1)
sample(1:10, 4)
# [1] 3 4 5 7
sample(1:10, 4)
# [1] 3 9 8 5
sample(letters, 5)
# [1] "q" "b" "e" "x" "p"
sample(1:10)     # permutation
# [1]  9 10  3  1  7  4  8  6  5  2
sample(1:10, replace = TRUE)
# [1]  9  9  7  9  5  7  4  4 10  8
```

Simulation

Summary:

- Drawing samples from specific probability can be done with `r`* functions.
- Standard distributions are built in: normal, Poisson, binomial, exponential, gamma etc.
- The `sample` function can be used to draw random samples from arbitrary vectors.
- Setting the random number generator seed via `set.seed` is critical for reproducibility.

R profiler

Why is my code so slow?

- Profiling is a systematic way to examine how much time is spent in different parts of a program.
- Useful when trying to optimize your code.
- Often code runs fine once, but what if you have to put it in a loop for 1000 iterations? Is it still fast enough?
- Profiling is better than guessing.

On optimizing your code

- Getting biggest impact on speeding up code depends on knowing where the code spends most of its time.
- This cannot be done without performance analysis or profiling.

> "We should forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil". (Donald Knuth)

General principles of optimization

- Design first, then optimize.
- Remember: "premature optimization is the root of all evil".
- Measure (collect data), don't guess.
- If you're going to be a scientist, you need to apply the same principles here.

Using `system.time`

- Takes an arbitrary R expression as input (can be wrapped in curly braces) and returns the amount of time taken to evaluate the expression.
- Computes the time (in seconds) needed to execute an expression.
  - If there is an error, gives the time until the error occurred.
- Returns an object of class `proc_time`.
  - `user_time`: time charged to the CPU(s) for this expression.
  - `elapsed_time`: "wall clock" time.
- Usually, the user time and elapsed time are relatively close, for straight computing tasks.
- Elapsed time may be *greater than* user time if the CPU spends a lot of time waiting around.
- Elapsed time may be *smaller than* user time if your machine has multiples cores/processors (and is capable of using them).
  - Multi-threaded BLAS libraries (`vecLib`/`Accelerate`, `ATLAS`, `ACML`, `MKL`).
  - Parallel processing via the `parallel` package.

```
# Elapsed time > user time
system.time(readLines("http://www.jhsph.edu"))
#     user     system   elapsed
#     0.14       0.06      4.16

# Elapsed time < user time
hilbert <- function(n) {
    i <- 1:n
    1 / outer(i - 1, i, "+")
}
x <- hilbert(1000)
system.time(svd(x))
#     user     system   elapsed
#     2.06       0.24      1.43
```

Timing longer expressions

```r
system.time({
    n <- 1000
    r <- numeric(n)
    for (i in 1:n) {
        x <- rnorm(n)
        r[i] <- mean(x)
    }
})
#      user     system    elapsed
#      0.07       0.00       0.07
```

Beyond `system.time`

- Using `system.time()` allows you to test certain functions or code blocks to see if they are taking excessive amount of time.
- Assumes you already know where the problem is and call `system.time()` on it.
- What if you don't know where to start?

The R profiler

- The `Rprof()` function starts the profiler in R.
  - R must be compiled with profiler support (but this is usually the case).
- The `summaryRprof()` function summarizes the output from `Rprof()` (otherwise, it is not readable).
  - There are two methods for normalizing the data:
    - `by.total` divides the time spent 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.
- DO NOT use `system.time()` and `Rprof()` together or you will be sad.
- `Rprof()` keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spent in each function.
- Default sampling interval is 0.02 seconds.
- NOTE: if your code runs very quickly, the profiler is not useful, but then you probably don't need it in that case.

Programming Assignment codes here.