notes_week4

Derek Hall

4/7/2020

The *str* function

Compactly describes contant of an object. One line per object. Great for nested lists.

```
x <- rnorm(100, 2, 4)

print(summary(x), str(x))
```

```
##  num [1:100] -1.662 -0.648 1.255 -0.83 5.344 ...
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -7.338276 -1.028429  1.432706  1.633641  4.334383  9.660012
```

Generating Random Numbers

Functions for generating random numbers: - *rnorm*: generates random Normal variates with a given mean and sd
- *dnorm*: evaluates the Normal probability density (with a given mean + sd) at a point (or vector of points)
- *pnorm*: evaluates the cumulative distribution function for a Normal distribution
- *rpois*: generate random Poisson variates with a given rate

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

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

Setting the random number seed with *set.seed* ensures **reproducibility**:

```
set.seed(1)
x <- rnorm(5)

y <- rnorm(5)

set.seed(1)
z <- rnorm(5)

matrix(data = c(x,y,z), nrow = 5, ncol = 3, dimnames = list(c(1:5), c("x", "y", "z")))
```

```
##            x          y          z
## 1 -0.6264538 -0.8204684 -0.6264538
```

```
## 2  0.1836433  0.4874291  0.1836433
## 3 -0.8356286  0.7383247 -0.8356286
## 4  1.5952808  0.5757814  1.5952808
## 5  0.3295078 -0.3053884  0.3295078
```

Note: random number generation on a computer is not actually random, it just appears random (pseudorandom).

Generating Random Numbers from a Linear Model

Suppose we want to simulate data from a model (e.g. linear model).

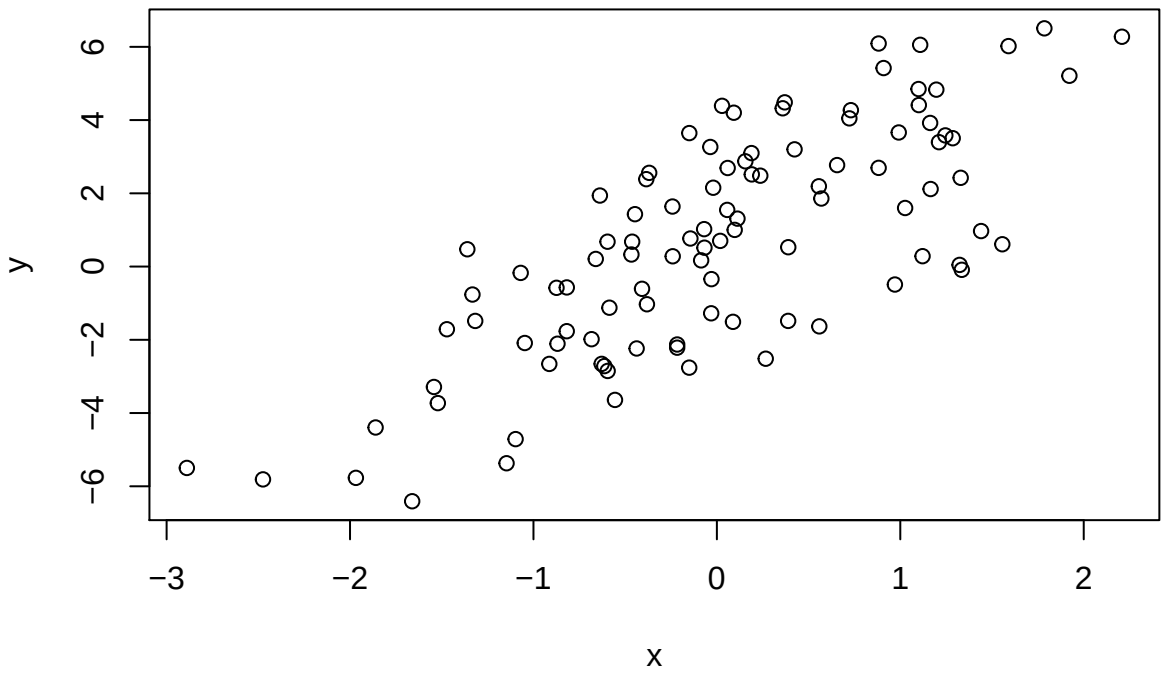
$y = b + mx + e$

- e = random noise from normal distribution (0,2).
- b = intercept of 0.5.
- m = slope of 2.
- x = normal distribution of data (0,1)

```
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
```

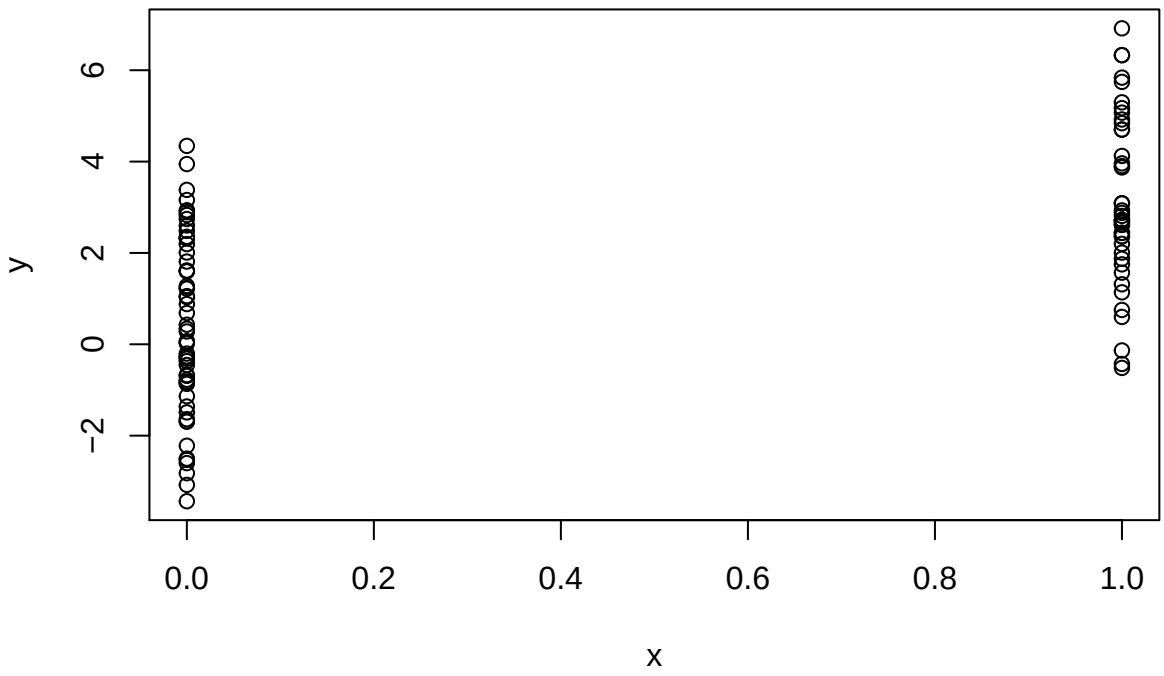
```
plot(x, y)
```



Note: Don't forget to set the seed!

Can use the *binom* function to get a binomial distrubtion

```
set.seed(20)
x <- rbinom(100, 1, 0.5)
e <- rnorm(100, 0 , 2)
y <- 0.5 + 2 * x + e
plot(x,y)
```



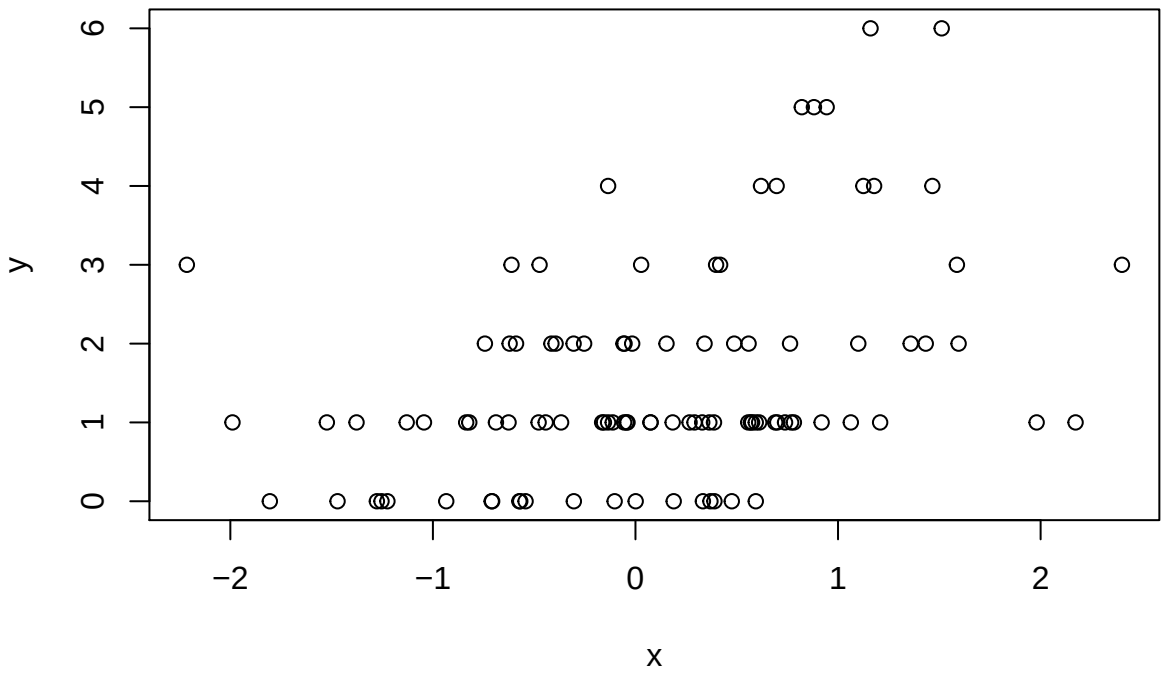
What about Poisson?

Y ~ Poisson(mu)

log(mu) = b + m * x

and b = 0.5 and m = 0.3.

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



Simulating random sampling

The *sample* function allows you to draw randomly from a given object.

```
set.seed(1)
sample(1:10, 4)
```

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

If you don't give it a sampling number, it generates a permutation:

```
sample(1:10)
```

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

Replacement puts the number back in the "pool" after it has been selected.

```
sample(1:10, replace = TRUE)
```

```
##  [1] 10  6 10  7  9  5  5  9  9  5
```

R Profiler

A tool to systematically assess how much time is spent on different parts of a program. Useful for trying to optimize code.

Don't try to optimize code from the start. Focus on getting the code running, then worry about optimizing it.

Don't guess at optimization. Collect data through profiling!

system.time()

system.time() evaluates an expression and then reports the time it takes to evaluate or to hit an error (in seconds).

- user time = the time the computer spends, the amount of time the CPU is being used.
- elapsed = the IRL time spent.

Elapsed > User when the CPU spends a lot of time waiting.

Elapsed < User when the computer analyzes in parallel (e.g. using multiple codes)

An easy tool to assess issues when you already know where to look.

Rprof()

Rprof() and *summaryRprof()*. Summary function makes the output readable.

DO NOT use *Rprof()* and *system.time()* together.

Rprof() keeps track of the function call stack at regular intervals and tabulates how much time is spent on each function. The default sampling time is 0.02 seconds

Two methods for normalizing Rprof data:

- *by.total* divides the time spent by the total run time.
- *by.self* does the same thing but first subtracts the time spent on the functions above in the call stack.