RHUL Psychology Statistical modelling notebook

Matteo Lisi

2022-06-29

Contents

Chapter 1

About

This online book is created and maintained by Matteo Lisi and is meant to be a shared resource for staff and students at the Department of Psychology of Royal Holloway, University of London. It will contain a miscellanous set of tutorial, examples, case studies, workshops materials and any other useful material related to data analysis and modelling. These will be added and revised over time, based on the most common questions and requests that I receive.

This is a work in progress and may contain imprecisions and typos. If you spot any please let me know at matteo.lisi [at] rhul.ac.uk. The materials that will be included builds upon and draw from existing literature on statistics and modelling. I will endeavor to properly cite existing books and papers; but if any author feels that I have not given them fair acknowledgement, please let me know and I will make amend.

Chapter 2

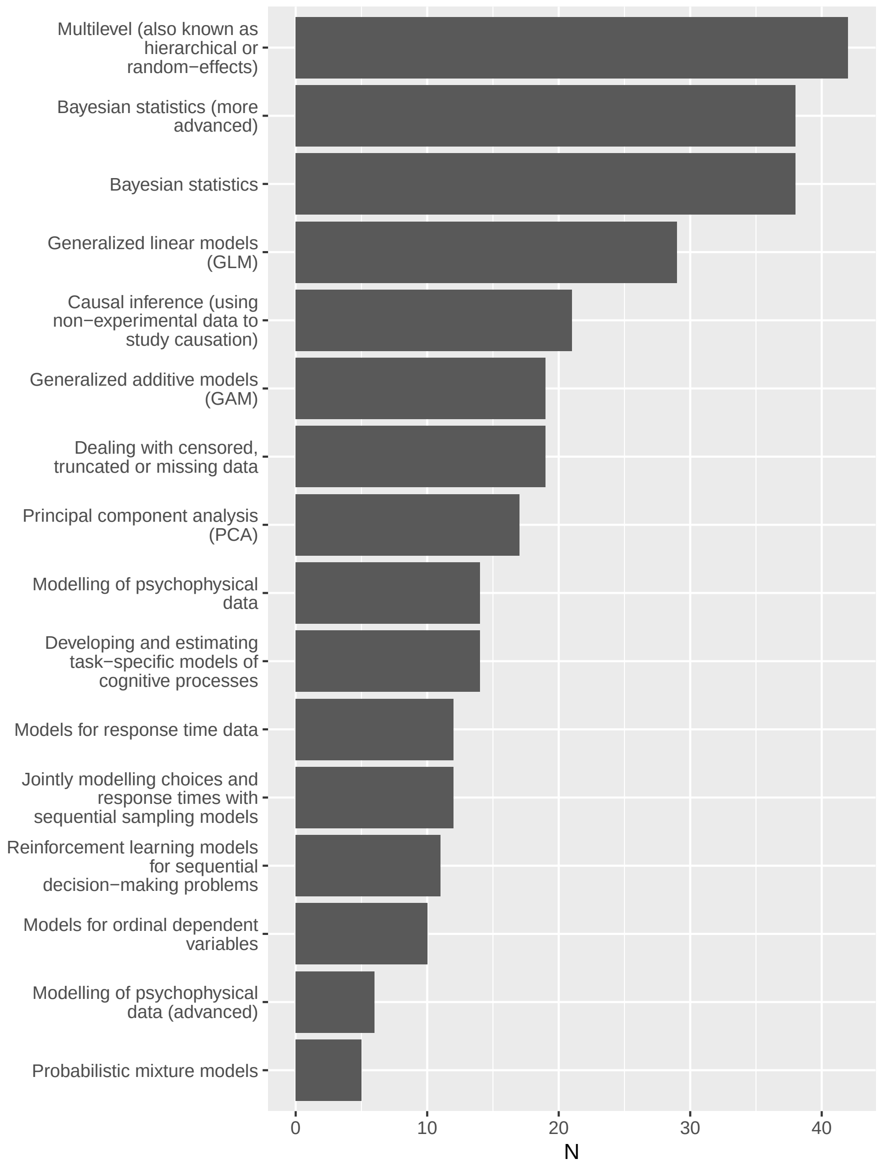
Departmental survey about statistical methods

I used an anonymous survey to ask colleagues some questions about which topics may be more interesting or useful in their research.

2.1 March 2022

2.1.1 Question 1

In the first question people indicated topics of interests. The winner are multi-level models, followed closely by Bayesian statistics.



There were some additional suggestions.

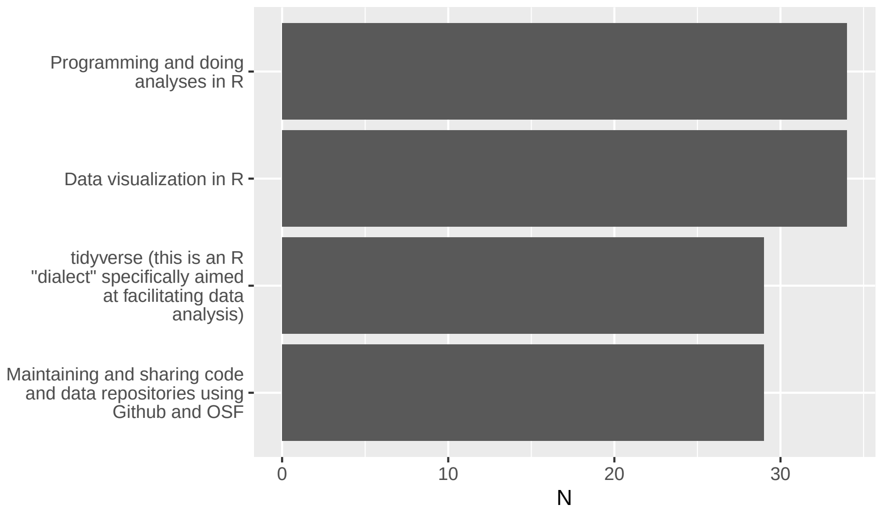
```
#> [1] "power analyses using Shiny apps"
#> [2] "agent-based models"
#> [3] "this may be covered in the above, but approaches to analysing experience sampl
#> [4] "Methods for longitudinal analyses"
#> [5] "Network modelling"
#> [6] "Neural networks, Markov processes"
#> [7] "Random forests and related"
#> [8] "causal modelling using regression models - path models etc"
#> [9] "prediction modelling"
```

A few other topics were mentioned in the comment section:

- Shiny apps
- Network modelling
- Longitudinal analyses
- Random forests
- Neural network

2.1.2 Question 2

Here people indicated their interest for topics related to data analysis.

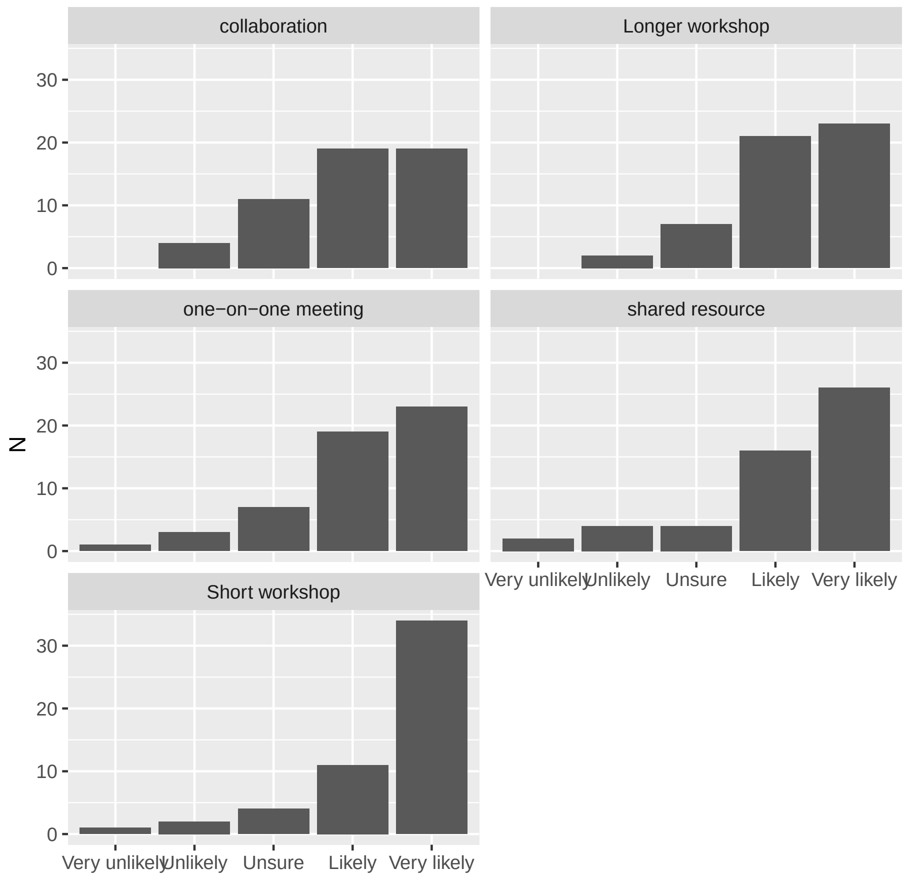


Other things mentioned in the comments were:

- SPM
- Docker
- Python

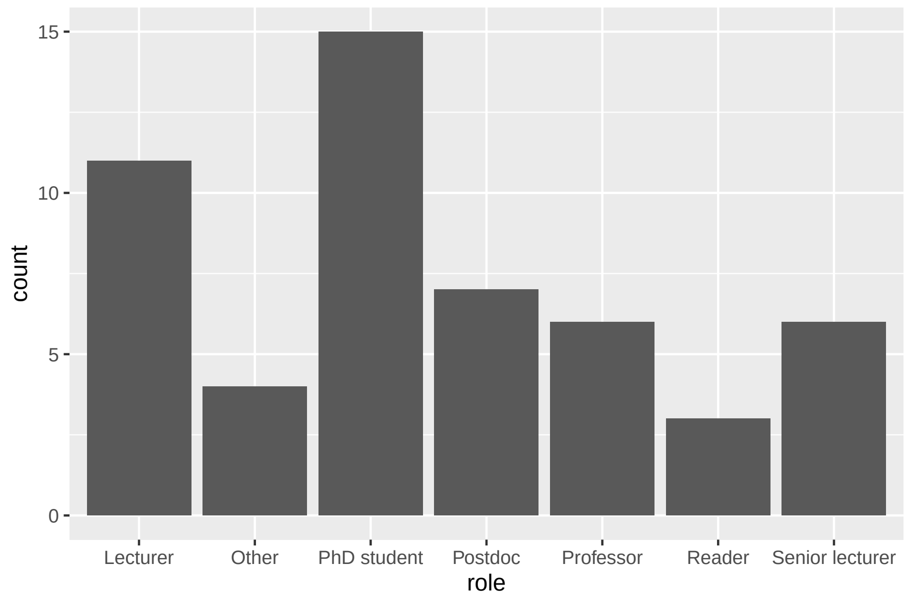
2.1.3 Question 4

This question was about likelihood of using different formats of support



2.1.4 Respondents' status

The final questions asked about the status / career level.


count
15
10
5
0
Lecturer
Other
PhD student
Postdoc
Professor
Reader
Senior lecturer
role

Chapter 3

Introduction to R

Most of the practical statistical tutorials and recipes in this book use the software R, so this section provides some introduction to R for the uninitiated.

3.1 Installing R

The base R system can be downloaded at the following link, which provides installers for both Windows, Mac and Linux:

https://cran.rstudio.com/

In addition to the base R system, it is useful to have also R-studio, which is an IDE (Integrated Development Environment) for R, and provides both an editor, a graphical interface and much more. It can be downloaded from:

https://www.rstudio.com/products/rstudio/download/

3.2 First steps

R is a programming language and free software environment for statistical computing and graphics. It is an *interpreted language*, which means that to give instructions to the computer you do not have to compile it first in machine language, everything is done 'on the fly' through a command line interpreter, e.g. if you type `2+2` in the command line R, the computer will reply with the answer (try this on your computer):

```
2+2
#> [1] 4
```

Typically the normal workflow involve writing and saving a series of instructions in a *script* file (usually saved with the `.R` extension), which can be executed (either step by step or all at once). Since all steps of the analyes are documented in the script, this makes them transparent and reproducible.

In an R script you can use the `#` sign to add comments, so that you and others can understand what the R code is about. Commented lines are ignored by R, so they will not influence your result. See the next example:

```
# calculate 3 + 4
3 + 4
#> [1] 7
```

3.2.1 Arithmetic with R

In its most basic form, R can be used as a simple calculator. Consider the following arithmetic operators:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Exponentiation: `^`
- Modulo: `%%`

The last two might need some explaining:

The ^ operator raises the number to its left to the power of the number to its right: for example `3^2` is 9.

The modulo returns the remainder of the division of the number to the left by the number on its right, for example 5 modulo 3 (or `5 %% 3`) is 2.

3.2.2 Variable assignment

A basic concept in programming (statistical or not) is called a *variable*.

A variable allows you to store a value (e.g. 2) or an object (e.g. a function description) in R. You can then later use this variable's name to easily access the value or the object that is stored within this variable.

You can assign a value `2` to a variable `my_var` with the command

```
my_var <- 2
```

Note that you would have obtained the same result using:

```
2 -> my_var
```

that is, the *assignment operator* works in both directions `<-` and `->`.

The variable can then be used in any computation, for example:

```
my_var + 2
#> [1] 4
```

3.2.3 Basic data types in R

Variables can be of many types, not just numerical values. For example, they can contain *text* values (e.g. a string of characters). Arithmetic operators such as `+` do no work with these. If you tried to apply them characters R will give you an error message.

```
# Assign a value to the variable apples
apples <- 5

# Assign a text value
oranges <- "six"

#
apples + oranges
#> Error in apples + oranges: non-numeric argument to binary operator
```

In fact R works with numerous data types, and some of these are not numerical (so they can't be added, subtracted, etc.). Some of the most basic types to get started are:

- Decimal values like 4.5 are called numerics.
- Natural numbers like 4 are called integers. Integers are also numerics.
- Boolean values (`TRUE` or `FALSE`, abbreviated `T` and `F`) are called logical[1].
- Text (or string) values are called characters.

3.2.4 Vectors and other data types

Additionally, the simple data types listed above can be combined in more complex 'objects' that can comprise several values. For example, we can obtain a *vector* by concatenating values using the function `c()`. This can be applied both on numerical or character data types, e.g.

[1]Note that you can add or multiply logical Boolean values: internally `FALSE` are treated as zeroes, and `TRUE` as ones.

```
some_numbers <- c(4,87,10, 0.5, -6)
some_numbers
#> [1]  4.0 87.0 10.0  0.5 -6.0

my_modules <- c("PS115", "PS509", "PS300", "PS938", "PS9457")
my_modules
#> [1] "PS115"  "PS509"  "PS300"  "PS938"  "PS9457"
```

There are some special handy functions to create specific types of vectors, such as *sequences* (using the function `seq()` or the operator `:`)

```
x <- seq(from = -10, to = 10, by = 2)
x
#>  [1] -10  -8  -6  -4  -2   0   2   4   6   8  10

y <- seq(-0, 1, 0.1)
y
#>  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

z <- 1:5
z
#> [1] 1 2 3 4 5
```

Another useful type of vector can be obtained by *repetition* of elements, and this can be numerical, character, or even applied to other vectors

```
rep(3, 5)
#> [1] 3 3 3 3 3

x <- 1:3
rep(x, 4)
#>  [1] 1 2 3 1 2 3 1 2 3 1 2 3

rep(c("leo the cat", "daisy the dog"), 2)
#> [1] "leo the cat"   "daisy the dog" "leo the cat"
#> [4] "daisy the dog"
```

We can combine vectors of different types into a *data frame*, one of the most useful ways of storing data in R. Let's say we have 3 vectors:

```
# create a numeric vector
a <- c(0, NA, 2:4)  # NA means not available

# create a character vector
b <-  c("PS115", "PS509", "PS300", "PS938", "PS9457")

# create a logical vector
```

```
c <- c(TRUE, FALSE, TRUE, FALSE, FALSE)  # must all be caps!
```

we can combine them into a data.frame using:

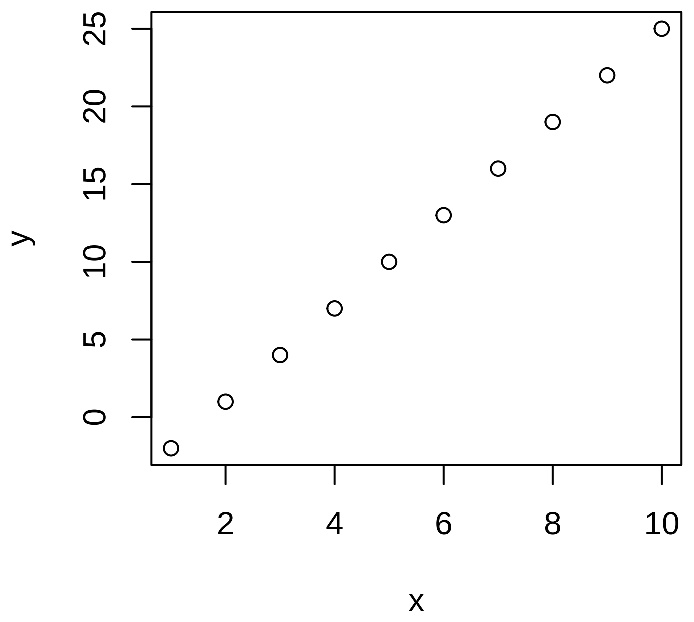
```
# create a data frame with the vectors a, b,and c that we just created
my_dataframe <- data.frame(a,b,c)

# we could also change the column names (currently they are a, b, c)
colnames(my_dataframe) <- c("some_numbers", "my_modules", "logical_values")

# now let's have a look at it
my_dataframe
#>   some_numbers my_modules logical_values
#> 1            0      PS115           TRUE
#> 2           NA      PS509          FALSE
#> 3            2      PS300           TRUE
#> 4            3      PS938          FALSE
#> 5            4     PS9457          FALSE
```

Although note that in most cases we would probably import a dataframe from an external data file, for example using the functions `read.table` or `read.csv`.

3.2.5 Basic plotting in R

We can create plots using the function `plot()`. For example:

```
x = 1:10
y = 3*x - 5
plot(x, y)
```

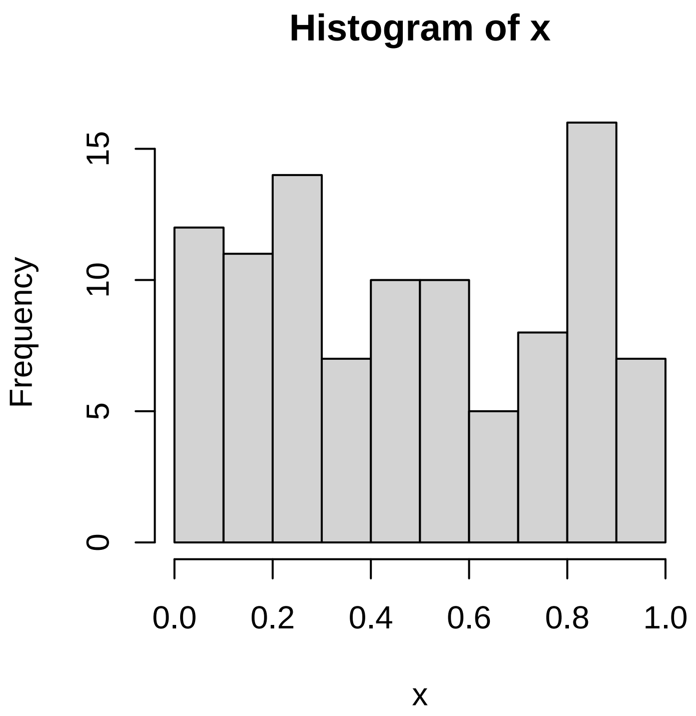


3.2.6 Other operations

3.2.6.1 Random number generation

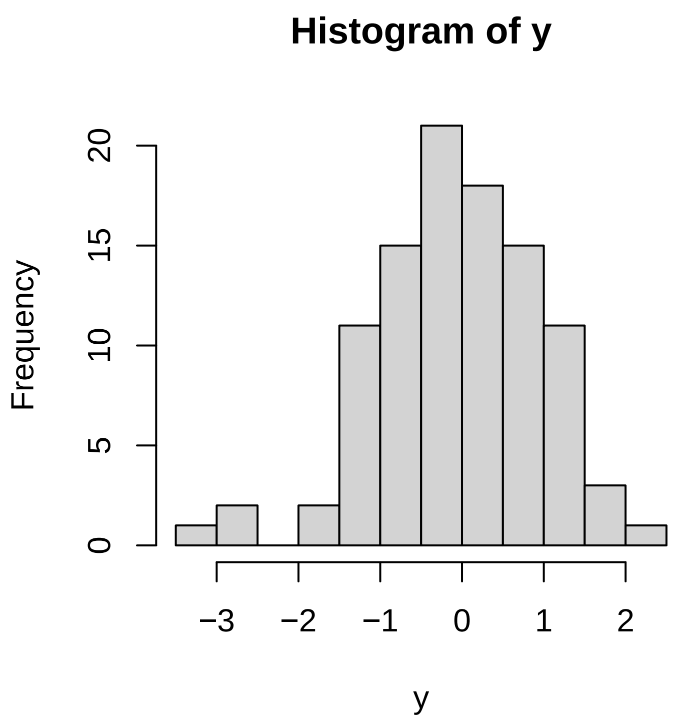
Generate uniformly distributed random numbers (function `runif()`)

```
x <- runif(100, min = 0, max = 1)
hist(x)
```



Generate numbers from a normal distribution

```
y <- rnorm(100, mean = 0, sd = 1)
hist(y)
```



3.2.7 Getting help

R has a lot of functions, and extra packages that can provides even more. It may seem a bit overwhelming, but it is very easy to get help about how to use a function: just type in a question mark, followed by the name of the function. For example, to see the help of the function we used above to generate the histogram, type

```
?hist
```

3.3 Resources for learning R

There is plenty of resources on the web to learn R. I will recommend a couple that I think are particularly well-done and useful:

- Software Carpentry tutorials on R for Reproducible Scientific Analysis
- The free book Learning Statistics with R by Danielle Navarro

Chapter 4

Linear models

This section will provide some worked examples of how to do analyses in R.

4.1 Simple linear regression

In this example[1] we will see how to import data into R and perform a simple linear regression analysis.

According to the standard big-bang model, the universe expands uniformly and locally, according to Hubble's law(**?**)

$$\text{velocity} = \beta \times \text{distance}$$

where velocity and distance are the relative velocity and distance of a galaxy, respectively; and β is "Hubble's constant"[2]. Note that this is a simple linear equation, in which β indicate how much the variable velocity changes for each unitary increase in the variable distance.

According to this model β^{-1} gives the approximate age of the universe, but β is unknown and must somehow be estimated from observations of velocity and distance, made for a variety of galaxies at different distances from us. Luckily we have available data from the Hubble Space Telescope. Velocities are assessed by measuring the Doppler effect red shift in the spectrum of light that we receive from the Galaxies. Distance is estimated more indirectly, by using the discovery that in certain class of stars (Cepheids), which display fluctuations in diameter and temperature over a stable period, there is a systematic relationship between the period and their luminosity.

[1]Taken from Simon Wood's book on GAM(**?**).

[2]Note the Hubble "constant" is a constant only in space, not in time

We can load a dataset of measurements from the Hubble Space Telescope in R using the following code

```
d <- read.table(file="https://raw.githubusercontent.com/mattelisi/RHUL-stats/main/data/
                header=T)
```

`read.table` is a generic function to import dataset in text files (e.g. .csv files) into R. We use the argument `header=T` to specify that the first line of the dataset gives the names of the columns. Note that the argument `file` here is a URL, but it could be also a path to a file in our local folder. To see the help of this function, and what other arguments and features are available type `?read.table` in the R command line.

We can use the command `str()` to examine what we have imported

```
str(d)
#> 'data.frame':    24 obs. of  3 variables:
#>  $ Galaxy  : chr  "NGC0300" "NGC0925" "NGC1326A" "NGC1365" ...
#>  $ velocity: int  133 664 1794 1594 1473 278 714 882 80 772 ...
#>  $ distance: num  2 9.16 16.14 17.95 21.88 ...
```

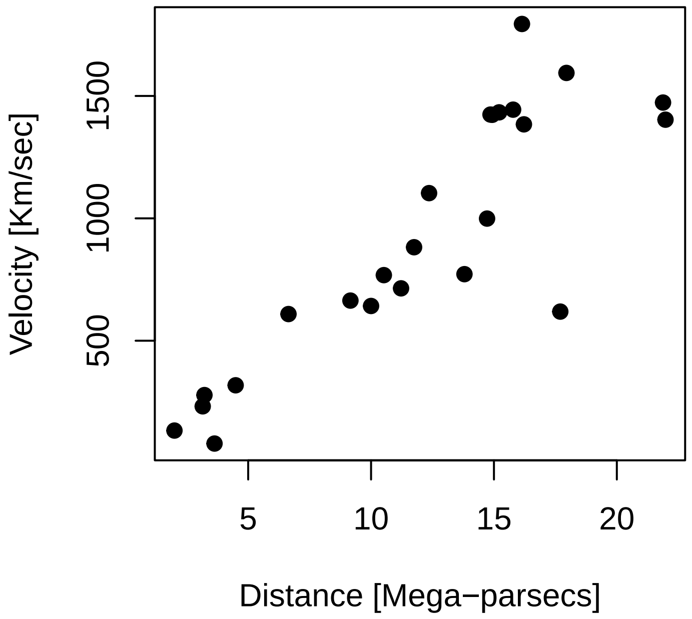
This tells us that our data frame has 3 variables:

- `Galaxy`, the 'names' of the galaxies in the dataset
- `velocity`, their relative velocity in Km/sec
- `distance`, their distance expressed in Mega-parsecs[3]

We can plot[4] them using the following code:

```
plot(d$distance, # indicate which variable on X axis
     d$velocity, # indicate which variable on Y axis
     xlab="Distance [Mega-parsecs]",
     ylab="Velocity [Km/sec]",
     pch=19) # set the type of point
```

[3] 1Mega-parsec $= 3.09 \times 10^{19}$Km
[4] See `?plot` for more info about how to customize plots in R.



It is clear, from the figure, that the observed data do not follow Hubble's law exactly, but given the how these measurements were obtained (there is uncertainty about the true values of the distance and velocities) it would be surprising if they did. Given the apparent variability, what can be inferred from these data? In particular:

1. what value of β is most consistent with the data?
2. what range of β values is consistent with the data?

In order to make inferences we make some assumptions about the nature of the measurement noise. Specifically, we assume that measurements errors are well-characterized by a Gaussian distribution. This result in the following model:

$$y = \beta x + \epsilon$$
$$\epsilon \sim \mathcal{N}\left(0, \sigma_\epsilon^2\right)$$

which is essentially a linear regression but without the intercept: that is, whereas normally a linear regression model include an additive term that is not multiplied with the predictor (as in $y = \beta_0 + \beta_1 x + \epsilon$), which gives the expected value of the dependent variable when all predictors are set to zero, in this case the theory tells us we can assume the intercept (the term β_0) is zero and we can ignore it.

We can fit the model with the function `lm` in R. Note that to tell R that I don't want to fit the intercept, I include in the formula the term `0 +` - this essentially tells R that the intercept term is set to zero[5]

```
hub.m <- lm(velocity ~ 0 + distance, d)
summary(hub.m)
#> 
#> Call:
#> lm(formula = velocity ~ 0 + distance, data = d)
#> 
#> Residuals:
#>    Min      1Q  Median      3Q     Max 
#> -736.5 -132.5   -19.0   172.2   558.0 
#> 
#> Coefficients:
#>          Estimate Std. Error t value Pr(>|t|)    
#> distance   76.581      3.965   19.32 1.03e-15 ***
#> ---
#> Signif. codes:  
#> 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#> 
#> Residual standard error: 258.9 on 23 degrees of freedom
#> Multiple R-squared:  0.9419, Adjusted R-squared:  0.9394 
#> F-statistic: 373.1 on 1 and 23 DF,  p-value: 1.032e-15
```

So, based on this data, **our estimate of the Hubble constant is 76.58 with a standard error of 3.96.** The standard error - which is the standard deviation of the sampling distribution of our estimates - gives an ideas of the range of values that is compatible with our data and could be used to compute a confidence intervals (roughly, we would expect that the 'true' values of the parameters lies in the interval defined by $\pm$ 2 standard errors 95% of the times).

So, how old is the universe?

The Hubble constant estimate have units of $\frac{\text{Km/sec}}{\text{Mega-parsecs}}$. A Mega-parsecs is 3.09×10^{19}Km, so we divide our estimate of $\hat{\beta}$ by this amount. The reciprocal of $\hat{\beta}$ then gives the approximate age of the universe (in seconds). In R we can calculate it (in years) as follow

```
# transform in Km
hubble.const <- coef(hub.m)/(3.09 * 10^(19))

# invert to get age in seconds
age <- 1/hubble.const
```

[5] A similar results would have been obtained using the notation `velocity ~ -1 + distance`.

```r
# use unname() to avoid carrying over
# the label "distance" from the model
age <- unname(age)

# transform age in years
age <- age/(60^2 * 24 * 365)

# age in billion years
age/10^9
#> [1] 12.79469
```

giving an estimate of about 13 billion years.

Chapter 5

Models for count data

```
#> -- Attaching packages ------------------- tidyverse 1.3.1 --
#> v ggplot2 3.3.6     v purrr   0.3.4
#> v tibble  3.1.7     v dplyr   1.0.9
#> v tidyr   1.2.0     v stringr 1.4.0
#> v readr   2.1.2     v forcats 0.5.1
#> -- Conflicts --------------------- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

This section will provide some examples of models that can deal with *count* data. Typically, count data occurs when the dependent variable is the counted number of occurrences of an event - for example the number of patients arriving in an emergency department (A&E) in a given time of the day - e.g. between 10:00 and 11:00. In this case, the dependent variable (the number of patients) has several characteristics that make it unsuitable for analysis with standard linear models such as linear regression: their distribution is discrete, composed only of non-negative integers, and is often positively skewed, with many observations having a value of 0.

Another characteristic is that the variance of the observations (e.g. the variance of the number of counts across observations within the same condition) increases with their expected value (e.g. the average number of counts for that condition)[1].

[1]This makes sense if you think that when the expected number of counts is very low, say ≈ 1, there cannot be many observations with very high counts - otherwise their average wouldn't be as low (recall that counts are strictly non-negative). In other words, the variance must be low when the average is also low.

5.1 Poisson model

The simplest model that account for the characteristics mentioned above is a generalized linear model that assume a dependent variable with a Poisson distribution. The Poisson distribution has a single free parameter, usually notated with λ, which gives both the expected value (the mean) and the variance of the count variable. In fact it assumes that the variance has the same value as the mean.

Formally, a Poisson model is usually formulated as follow: let $y = (y_1, \dots, y_n)$ be the dependent variable, consisting of counts (non-negative integers) and x the independent variable (predictor). Then

$$y_i \sim \text{Poisson}\left(\lambda_i\right)$$

where[2]

$$\log(\lambda_i) = \beta_0 + \beta_1 x_i$$

or alternatively

$$\lambda_i = e^{\beta_0 + \beta_1 x_i}$$

Which indicates that we can use the exponential function (in R the function `exp()`) to calculate the predicted values of the mean (more precisely, the expected value, $\mathbb{E}(y)$) and variance ($Var(y)$) of the dependent variable. This relies on the property of the Poisson distribution[3]

$$y_i \sim \text{Poisson}\left(\lambda\right) \implies \mathbb{E}(y) = Var(y) = \lambda$$

However, in practice, data often do not conform to the constraint of having identical mean and variance. Often the observed variance of the count is higher than what predicted according to the Poisson model (we say in this case that the data is *over-dispersed*).

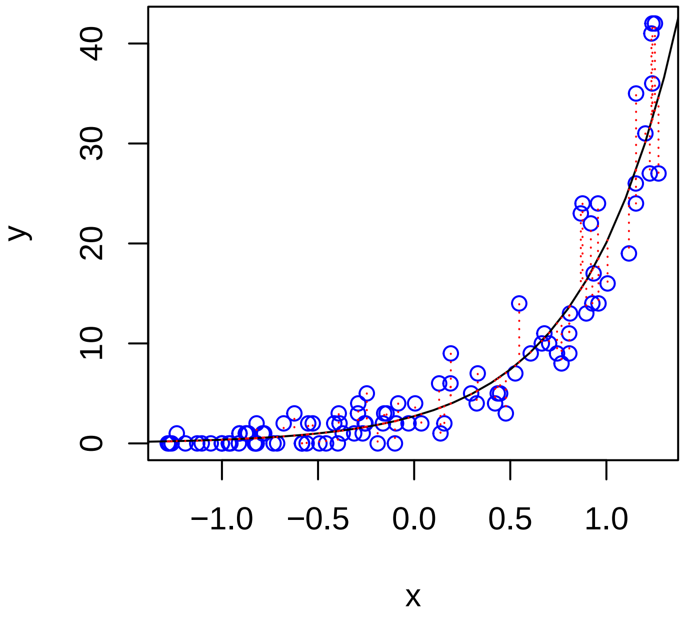
How does over-dispersion look like?

The type of data expected under a Poisson model is illustrated in the figure below, which shows 100 datapoints simulated from the model $y_i \sim \text{Poisson}\left(e^{1+2x_i}\right)$. The vertical deviations of the datapoints from the line are consistent with the property of the Poisson distribution that the variance of hte count has the same value as their expected value, formally, $Var(y) = \mathbb{E}(y)$).

[2]This is the most common formulation, and is sometime referred to as log 'link' function, to indicate the fact that we have a function (the logarithm function) that 'link' the linear part of the model (the linear combination of the independent variables, here $\beta_0 + \beta_1 x_i$) with the parameter of the distribution of the dependent variable (here the Poisson rate parameter λ). This is a common component to all generalized linear models, for example for the logistic regression we have a 'logit' link function - the quantile function of the standard logistic distribution - that link the linear predictor part with the parameter p of the binomial distribution.

[3]The symbol $\implies$ is used to denote logical implication, e.g. $(A = B) \implies (B = A)$ - symmetry of logical equivalence.

```
set.seed(2)
n <- 100
x <- runif(n, -1.3, 1.3)
a <- 1
b <- 2
linpred <- a + b*x # linear predictor part
y <- rpois(n, exp(linpred)) # simulate Poisson observations

plot(x,y,col="blue") # plot
x_ <- seq(-2,2,0.1)
lines(x_, exp(a+b*x_))
segments(x,exp(a+b*x),x,y, lwd=1,lty=3, col="red")
```



We can adjust the code above to simulate the same data with some degree of over-dispersion, using a negative binomial distribution, for different values of the precision parameter theta (θ), which regulate the degree of overdispersion. Importantly, these datapoints are simulated assuming the same function fo the average (expected) number of counts (same also as the previous figure), they just differe in the amount of overdispersion relative to a Poissone model. Note