

Statistical Principles & Experimental Design

Seminar 1a: R Basics & Random Samples

Solutions


matthias.kuhn@tu-dresden.de

Institute for Medical Informatics and Biometry (IMB)
Faculty of Medicine Carl Gustav Carus
Technische Universität Dresden

October 19, 2023 ^[39e18ee]
(solutions interspersed)

In this session we explore the R programming language. You learn what data types and structures are available in R. Later, you see an example how to explore the distribution of a measurement based on a random data sample.

For each session it is recommended that you start off with a new R script and to save it regularly to disk while working on it.

1. Start **RStudio**, look for the  icon.
2. Create a new R script in RStudio and save it to file already under the name – say – `StatPrinExpDsgn_seminar_R_basics.R` in a dedicated folder on your computer. Remember that you can execute portions of your script in RStudio via `Ctrl` + `↵`.

Conventions. In this and the following handouts we present R code in grey text boxes like the following:

```
t.test(x=c(1,2,3), y=c(2,3,4))  
plot(speed ~ dist, data=cars)  
image(volcano)
```

You can use these as a sample for your own scripts. R-*output* in the grey text boxes is indicated by two hashes `##` put in front.

Run them and try to understand what they do. For each of these functions please also take a look at the help page (e.g. by typing `?mean` in the R console) to get used to them. And what happens if you omit the function arguments and only call the function name, e.g. `order` without parentheses?

Own functions in R. You are not limited to the predefined functions. Functions are actually first-class objects in R. A simple example of a self-defined function in R is:

```
squ <- function(p) { p^2 }
```

By running this code you define a function called `squ` that calculates the square of the given argument. The last statement in the function code yields the return value of the function. You can also explicitly use `return()`-statements to exit the function. Since version 4.1, R provides the shorthand notation `\(p) {p^2}` for a function definition. Either way, you use your function `squ` like so:

```
squ(5)
squ(-2.3) # negative number
squ(-2:3) # vector of length 6
```

What happens if you apply your function `squ` on a character string, e.g. `squ("hallo")`? And what happens if you give no argument, `squ()`, or if you only type `squ`, without the round brackets?

Our function `squ` inherits the ability to work with whole vectors from the exponentiation operator `^`. It returns a vector of the same length as the input vector where each element is squared.

Running `squ` on a character argument returns an error that gets propagated from the exponentiation operator `^`, it expects a numeric argument. Likewise, if you miss an argument that has no default value defined you get an error. You would provide a default argument value by using something like `function(p=5)` in the function definition of `squ`.

3 Further R material

Obviously, we have just scratched the surface of R and there are plenty of R-tutorials and reference books out there. To name just a few freely accessible in the internet:

- The official R-manual <https://cran.r-project.org/manuals.html>

- The R-help mailing list <https://www.r-project.org/mail.html>
- The online site to the book *R In Action* <https://www.statmethods.net>
- Federated R blog <https://www.r-bloggers.com>
- An in-depth textbook on R <https://adv-r.hadley.nz>. This website is very useful also as reference site where you can quickly dig deeper into your specific R-topic at hand.
- Tidyverse style guide for R programming <https://style.tidyverse.org>

4 Populations, random samples & measurements

In the first lecture, you have heard of the *two worlds of statistics*. First, there is the conceptual **population**, a set of things of interest like -say- *all* healthy murine (=from mouse) liver cells. A **random variable** X represents a *measurement of interest* which can be taken on each element in the population, say, the activity of the protein catalase, measured in enzyme units. The population is considered to be *homogeneous*, i.e. all measurements on the population follow the same distribution. Furthermore, in classical statistics, the measurements are assumed to be independent of each other and each measurement is modelled by a **theoretical distribution** like the normal distribution (with parameters expectation μ and variance σ^2).

But, the whole population is often intangible or too big to oversee. Therefore in statistics, one resorts to a finite **random sample** of size n taken from that (infinite) population in question. For a homogeneous population, the measurements $X_i, (i = 1, \dots, n)$ in the sample will be *identically distributed* and are assumed to be *independent* of each other. These two properties are often abbreviated as *iid*.

Realized measurements from a given, concrete sample are just n numbers but the fundamental theorem of statistics due to Glivenko-Cantelli guarantees that on principle a sample is a well suited device to let us learn ("*infer*") about the distribution of the measurement. The inference becomes stronger the larger the sample size n is.

1. The conceptual world: population & theoretical distribution

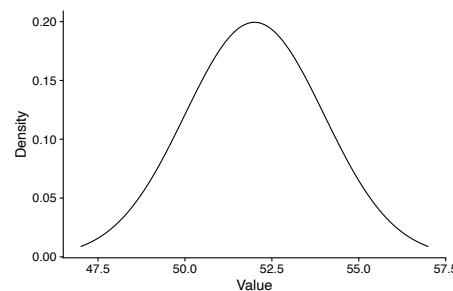
Imagine, the activity of the protein catalase in healthy murine liver cells (measured in enzyme units) under a set of certain fixed conditions is normally distributed with expectation $\mu = 52$ and variance $\sigma^2 = 4$. The normal distribution is a family of theoretical parametric distributions with two parameters μ and σ^2 . For given values for μ and σ^2 , one can calculate different properties of this theoretical distribution.

Density function for a normally distributed random variable X is

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

- Implement the normal density as an R function with three parameters for value x , mean μ and variance σ^2 !
- Test your density functions for the mentioned distribution of protein catalase. When assuming $\mu = 52$ and $\sigma^2 = 4$ you should get the following values depending on x :

Value	Density
47	0.00876
50	0.12099
52	0.19947
54	0.12099
55	0.06476



We implement the density function manually. Actually, it is already defined in R under the name `dnorm`. To plot a function with `ggplot2` we use the layer `geom_function`.

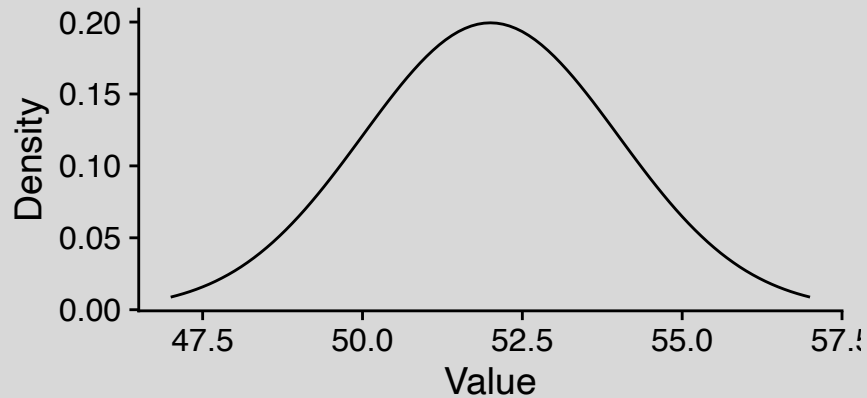
```
dens_norm <- function(x, m=52, v=4){
  1 / sqrt(2 * pi * v) * exp(-(x-m)^2 / (2 * v))
}
dens_norm(x=c(47, 50, 52, 54, 55))

## [1] 0.0087642 0.1209854 0.1994711 0.1209854 0.0647588

library("ggplot2")
ggplot() + xlim(47, 57) +
  geom_function(fun = dens_norm) +
  labs(x = 'Value', y = 'Density',
       title = 'Normal Probability Density',
       subtitle = 'with expected value 52 and variance 4')
```

Normal Probability Density

with expected value 52 and variance 4



- How would you calculate the probability to have a value of ≤ 54 ?

For a given value, the density of a continuous random variable quantifies how likely a realization actually is *in the neighbourhood* of that value. Hence, the probability to have a value ≤ 54 is the average of the density values below or equal of 54. For a continuous distribution like the normal distribution, this is the integral from $-\infty$ to 54. By the way, this is precisely the definition of the cumulative distribution function F_X .

- How would you calculate the probability to have a value between 48 and 56?

Accordingly, the probability for a finite region is expressed as the integral of the normal density over that region, in the given example from 48 to 56. This corresponds to the interval of $2 \times$ standard deviation below and above the mean.

- Can you do the actual calculations in R? *Hint:* Function `integrate`

The R-function `integrate` allows numeric integration of a one-dimensional function. Because integration of probability densities is often needed to calculate P-values there are already R-functions predefined for that. In our case the function is `pnorm`, `p` for probability and `norm` for normal distribution family.

```

integrate(f = dens_norm, lower = -Inf, upper = 54)

## 0.84134 with absolute error < 1.6e-09

# Or alternatively: use predefined function pnorm
pnorm(54, mean = 52, sd = 2)

## [1] 0.84134

# This is the probability to have a realization below or
# equal 54.

integrate(f = dens_norm, lower = 48, upper = 56)

## 0.9545 with absolute error < 1.8e-11

# Or alternatively: use predefined function pnorm
pnorm(56, mean = 52, sd = 2) - pnorm(48, mean = 52, sd = 2)

## [1] 0.9545

```

2. The sample world

Now, let's assume that you have a concrete random sample of these aforementioned measurements stemming from liver cells of different healthy mice. Actually, we can virtually generate a sample from a given theoretical distribution! For the normal distribution the relevant R function is `rnorm()` - the first letter `r` stands for random and means that you draw a *random* sample from that distribution. The following code draws a sample of size $n = 57$ from the aforementioned normal distribution. Please check the help page for `rnorm`. For reproducible results you can set the state of the pseudo-random generator via `set.seed`.

```

set.seed(12345)
x <- rnorm(57, mean=52, sd=2)

```

Now that we have a virtual random sample you can do **descriptive statistics** on it. In a real-world application you generally do *not* know the real underlying distribution in the population. Instead, you *estimate* and *infer* from the sample how the measurements

in the homogeneous population are distributed. The theoretic foundation is given by the *fundamental theorem of statistics* due to Gliwenko-Cantelli that guarantees that the empirical cumulative distribution function of the measurements in the random sample (understood as random variables) converges uniformly to the true distribution of the measurement as sample size n grows to infinity.

Important descriptive summary statistics of metric-scale measurements in a sample are:

1. **Arithmetic Mean** is the most often used measure of location (center)

$$\bar{x} = \frac{1}{n} \cdot \sum_{i=1}^n x_i$$

```
mean(x) # mean of values in vector x
mean(x, trim=0.1) #look up the meaning of parameter trim= in ?mean
```

We can verify („by hand“) that the mean function is really the arithmetic mean as defined in the lecture:

```
sum(x) / length(x)
```

2. **Median** - the middle value of your sample. It is a more robust variant of a location estimator compared to mean.

```
median(x)
```

3. **Empirical variance** (or: sample variance) is a measure of spread (also called scale) of the data

$$s^2 = \frac{1}{n-1} \cdot \sum_{i=1}^n (x_i - \bar{x})^2$$

```
var(x)
```

Like for the arithmetic mean, please verify by hand that this function really complies with the given formula.

We exploit that R is vectorized

```
myvar <- function(x) 1/(length(x)-1) * sum((x-mean(x))^2)
stopifnot( isTRUE(all.equal(var(x), myvar(x))) ) # check result
```

4. **Empirical standard deviation** s as square root of the sample variance, i.e. $s = \sqrt{s^2}$

```
sd(x)
```

5. To get a concise **summary** of your sample in the numeric vector **x** use the **summary** function:

```
summary(x)
```

6. To see the effect of the sample size n with respect to the descriptive statistics you can draw samples with increasing sample size and check what happens with the sample estimates for mean, median and variance. According to the Gliwenko-Cantelli theorem the sample estimators like mean and variance (understood as random variables) will converge uniformly to the underlying parameter values from the true distribution with increasing n . Do you observe this when you increase the sample size for concrete samples?

We write a simulation function which takes sample size n as first parameter and that calculates the empirical mean and variance. We observe that with increasing n the sample mean approaches $\mu = 52$ and sample variance approaches $\sigma^2 = 4$. To get reproducible results you need to set the pseudo random generator in R (which is relevant for each call to **rnorm**) with function **set.seed**. We provide a seed-parameter to our simulation function.

```
mySimulationFun <- function(n, mean=52, sd=2, seed=NULL){  
  #optionally set seed (if seed argument is appropriate)  
  if (! is.null(seed) && is.numeric(seed)) set.seed(seed)  
  x <- rnorm(n=n, mean=mean, sd=sd)  
  # return vector with sample mean and variance  
  c(mean = mean(x), var = var(x))  
}  
  
# simulations with increasing sample size, for a fixed random seed  
mySimulationFun(n = 10, seed = 987)  
mySimulationFun(n = 100, seed = 987)  
mySimulationFun(n = 1000, seed = 987)  
mySimulationFun(n = 10000, seed = 987)  
  
# alternatively, produce a list-object in one go,  
# see ?map from package purrr  
# purrr is part of tidyverse (like the packages 'dplyr' and 'ggplot2')  
library("purrr")  
purrr::map(10^(1:4), .f = mySimulationFun, seed = 987)
```

Congratulations, you have accomplished your first exercise in R!