Data Handling: Import, Cleaning and Visualisation

Lecture 10: Data Analysis and Basic Statistics with R

Prof. Dr. Ulrich Matter
(University of St.Gallen)

09/12/2021

1 Data analysis with R

In the first part of this lecture we take a look at some key functions for applied data analysis in R. At this point, we have already implemented the collecting/importing and cleaning of the raw data. The analysis part can be thought of as a collection of tasks with the aim of making sense of the data. In practice, this can be explorative (discovering interesting patterns in the data) or inductive (testing of a specific hypothesis). Moreover it typically involves both functions for actual statistical analysis as well as various functions to select, combine, filter, and aggregate data. Similar to the topic of data cleaning/preparation, covering all aspects of applied data analysis with R goes well beyond the scope of one lecture. The aim is thus to give a practical overview of some of the basic concepts and their corresponding R functions (here from `tidyverse`).

1.1 Merging datasets

The following two data sets contain data on persons' characteristics as well as their consumption spending. Both are cleaned datasets. But, for analysis purposes we have to combine the two datasets in one. There are several ways to do this in R but most commonly (for `data.frames` as well as `tibbles`) we can use the `merge()`-function.

```r
# load packages
library(tidyverse)

# initiate data frame on persons' personal spending
df_c <- data.frame(id = c(1:3,1:3),
                   money_spent= c(1000, 2000, 6000, 1500, 3000, 5500),
                   currency = c("CHF", "CHF", "USD", "EUR", "CHF", "USD"),
                   year=c(2017,2017,2017,2018,2018,2018))
df_c
```

```
##   id money_spent currency year
## 1  1        1000      CHF 2017
## 2  2        2000      CHF 2017
## 3  3        6000      USD 2017
## 4  1        1500      EUR 2018
## 5  2        3000      CHF 2018
## 6  3        5500      USD 2018
```

```
# initiate data frame on persons' characteristics
df_p <- data.frame(id = 1:4,
                   first_name = c("Anna", "Betty", "Claire", "Diane"),
                   profession = c("Economist", "Data Scientist",
                                  "Data Scientist", "Economist"))
df_p
```

```
##   id first_name     profession
## 1  1       Anna      Economist
## 2  2      Betty Data Scientist
## 3  3     Claire Data Scientist
## 4  4      Diane      Economist
```

Our aim is to compute the average spending by profession. Therefore, we want to link `money_spent` with `profession`. Both datasets contain a unique identifier `id` which we can use to link the observations via `merge()`.

```
df_merged <- merge(df_p, df_c, by="id")
df_merged
```

```
##   id first_name     profession money_spent currency year
## 1  1       Anna      Economist        1000      CHF 2017
## 2  1       Anna      Economist        1500      EUR 2018
## 3  2      Betty Data Scientist        2000      CHF 2017
## 4  2      Betty Data Scientist        3000      CHF 2018
## 5  3     Claire Data Scientist        6000      USD 2017
## 6  3     Claire Data Scientist        5500      USD 2018
```

Note how only the exact matches are merged. The observation of `"Diane"` is not part of the merged data frame because there is no corresponding row in `df_c` with her spending information. If for some reason we would like to have all persons in the merged dataset, we can specify the `merge()`-call accordingly:

```
df_merged2 <- merge(df_p, df_c, by="id", all = TRUE)
df_merged2
```

```
##   id first_name     profession money_spent currency year
## 1  1       Anna      Economist        1000      CHF 2017
## 2  1       Anna      Economist        1500      EUR 2018
## 3  2      Betty Data Scientist        2000      CHF 2017
## 4  2      Betty Data Scientist        3000      CHF 2018
## 5  3     Claire Data Scientist        6000      USD 2017
## 6  3     Claire Data Scientist        5500      USD 2018
## 7  4      Diane      Economist          NA     <NA>   NA
```

1.2 Selecting subsets

For the next steps of our analysis, we actually do not need to have all columns. Via the `select()`-function provided in `tidyverse` we can easily select the columns of interest

```
df_selection <- select(df_merged, id, year, money_spent, currency)
df_selection
```

```
##   id year money_spent currency
## 1  1 2017        1000      CHF
## 2  1 2018        1500      EUR
## 3  2 2017        2000      CHF
## 4  2 2018        3000      CHF
## 5  3 2017        6000      USD
## 6  3 2018        5500      USD
```

1.3 Filtering datasets

In a next step, we want to select only observations with specific characteristics. Say, we want to select only observations from 2018. Again there are several ways to do this in R but a most comfortable way is to use the `filter()` function provided in `tidyverse`.

```
filter(df_selection, year == 2018)
```

```
##   id year money_spent currency
## 1  1 2018        1500      EUR
## 2  2 2018        3000      CHF
## 3  3 2018        5500      USD
```

We can use several filtering conditions simultaneously:

```
filter(df_selection, year == 2018, money_spent < 5000, currency=="EUR")
```

```
##   id year money_spent currency
## 1  1 2018        1500      EUR
```

1.4 Mutating datasets

Before we compute aggregate statistics based on our selected dataset, we have to deal with the fact that the `money_spent`-variable is not really tidy. It does describe a characteristic of each observation but it is measured in different units (here, different currencies) across some of these observations. If the aim was to have a perfectly tidy dataset, we could address the issue with `spread()`. However, in this context it could be more helpful to add an additional variable/column with a normalized amount of money spent. That is, we want to have every value converted to one currency (given a certain exchange rate). In order to do so, we use the `mutate()` function (again provided in `tidyverse`).

First, we look up the USD/CHF and EUR/CHF exchange rates and add those as a variable (CHF/CHF exchange rates are equal to 1, of course).

```
exchange_rates <- data.frame(exchange_rate= c(0.9, 1, 1.2),
                             currency=c("USD", "CHF", "EUR"), stringsAsFactors = FALSE)
df_selection <- merge(df_selection, exchange_rates, by="currency")
```

Now we can define an additional variable with the money spent in CHF via `mutate()`:

```
df_mutated <- mutate(df_selection, money_spent_chf = money_spent * exchange_rate)
df_mutated
```

```
##   currency id year money_spent exchange_rate money_spent_chf
## 1      CHF  1 2017        1000           1.0            1000
## 2      CHF  2 2017        2000           1.0            2000
## 3      CHF  2 2018        3000           1.0            3000
## 4      EUR  1 2018        1500           1.2            1800
## 5      USD  3 2017        6000           0.9            5400
## 6      USD  3 2018        5500            0.9             4950
```

1.5 Aggregation and summary statistics

Now we can start analyzing the dataset. Quite typically, the first step of analyzing a dataset is to get an overview by computing some summary statistics. This helps to better understand the dataset at hand. Key summary statistics of the variables of interest are the mean, standard deviation, median, and number of observations. Together, they give a first idea of how the variables of interest are distributed.

As you know from previous lectures, R has several built-in functions that help us do this. In practice, these basic functions are often combined with functions implemented particularly for this step of the analysis, such as `summarise()` provided in `tidyverse`.

As a first output in our report we want to show the key characteristics of the spending data in one table.

```
summarise(df_mutated,
          mean = mean(money_spent_chf),
          standard_deviation = sd(money_spent_chf),
          median = median(money_spent_chf),
          N = n())
```

```
##   mean standard_deviation median N
## 1 3025           1788.785   2500 6
```

Moreover we can compute the same statistics grouped by certain observation characteristics. For example, we can compute the same summary statistics per year of observation.

```
by_year <- group_by(df_mutated, year)
summarise(by_year,
          mean = mean(money_spent_chf),
          standard_deviation = sd(money_spent_chf),
          median = median(money_spent_chf),
          N = n())
```

```
## # A tibble: 2 x 5
##    year  mean standard_deviation median     N
##   <dbl> <dbl>              <dbl>  <dbl> <int>
## 1  2017  2800              2307.   2000     3
## 2  2018  3250              1590.   3000     3
```

Alternatively to the more user-friendly (but less flexible) `summarise` function, we can use lower-level functions to compute aggregate statistics, provided in the basic R distribution. A good example for such a function is `sapply()`. In simple terms, `sapply()` takes a list as input and applies a function to the content of each element in this list (here: compute a statistic for each column). To illustrate this point, we load the already familiar `swiss` dataset.

```
# load data
data("swiss")
```

Now we want to compute the mean for each of the variables in this dataset. Technically speaking, a data frame is a list, where each list element is a column of the same length. Thus, we can use `sapply()` to 'apply' the function `mean()` to each of the columns in `swiss`.

```
sapply(swiss, mean)
```

```
##        Fertility      Agriculture      Examination        Education         Catholic
##         70.14255         50.65957         16.48936         10.97872         41.14383
## Infant.Mortality
##         19.94255
```

By default, `sapply()` returns a vector or a matrix.[1] We can get a very similar result by using `summarise()`. However, we would have to explicitly mention which variables we want as input.

```
summarise(swiss,
          Fertility = mean(Fertility),
          Agriculture = mean(Agriculture)) # etc.
```

```
##   Fertility Agriculture
## 1  70.14255    50.65957
```

2 Tutorial: Analise messy Excel sheets

The following tutorial is a (substantially) shortened and simplified version of Ista Zahn and Daina Bouquin's "Cleaning up messy data tutorial" (Harvard Datafest 2017). The aim of the tutorial is to clean up an Excel sheet provided by the UK Office of National Statistics that provides data on the most popular baby names in England and Wales in 2015. The dataset is stored in `data/2015boysnamesfinal.xlsx`

2.1 Preparatory steps

```
## SET UP -------------------
# load packages
library(tidyverse)
library(readxl)

# fix variables
INPUT_PATH <- "data/2015boysnamesfinal.xlsx"
```

Before diving into the data import and cleaning, it is helpful to first open the file in Excel. We notice a couple of things there: first, there are several sheets in this Excel-file. For this exercise, we only rely on the sheet called "Table 1". Second, in this sheet we notice intuitively some potential problems with importing this dataset due to the way the spreadsheet is organized. The actual data entries only start on row 7. These two issues can be taken into consideration directly when importing the data with `read_excel()`.

```
## LOAD/INSPECT DATA -----------------

# import the excel sheet
boys <- read_excel(INPUT_PATH, col_names = TRUE,
                   sheet = "Table 1", # the name of the sheet to be loaded into R
                   skip = 6 # skip the first 6 rows of the original sheet,
                   )
```

```
## New names:
## * Rank -> Rank...1
## * Name -> Name...2
## * Count -> Count...3
## * `since 2014` -> `since 2014...4`
## * `since 2005` -> `since 2005...5`
## * ...
```

[1]The related function `lapply()`, returns a list (see `lapply(swiss, mean)`).

```r
# inspect
boys
```

```
## # A tibble: 61 x 11
##    Rank...1 Name...2 Count...3 `since 2014...4` `since 2005...5` ...6  Rank...7 Name...8 Count...9
##    <chr>    <chr>        <dbl> <chr>            <chr>            <lgl> <chr>    <chr>        <dbl>
##  1 <NA>     <NA>            NA <NA>             <NA>             NA    <NA>     <NA>            NA
##  2 1        OLIVER        6941                  +4               NA    51       REUBEN        1188
##  3 2        JACK          5371                  -1               NA    52       HARLEY        1175
##  4 3        HARRY         5308                  +6               NA    53       LUCA          1167
##  5 4        GEORGE        4869 +3               +13              NA    54       MICHAEL       1165
##  6 5        JACOB         4850 -1               +16              NA    55       HUGO          1153
##  7 6        CHARLIE       4831 -1               +6               NA    56       LEWIS         1148
##  8 7        NOAH          4148 +4               +44              NA    57       FRANKIE       1112
##  9 8        WILLIAM       4083 +2                                NA    58       LUKE          1095
## 10 9        THOMAS        4075 -3               -6               NA    59       STANLEY       1078
## # ... with 51 more rows, and 2 more variables: `since 2014...10` <chr>, `since 2005...11` <chr>
```

Note that by default, `read_excel()` "repairs" the column names of imported datasets in order to make sure all columns do have unique names. For the moment we do not need to worry about the automatically assigned column names. However, some of the columns are not needed for analytics purposes. In addition, we note that some rows are empty (contain `NA` values). In a next step we thus *select* only those columns needed and *filter* incomplete observations out.

```r
# FILTER/CLEAN ---------------------------

# select columns
boys <- select(boys, Rank...1, Name...2, Count...3, Rank...7, Name...8, Count...9)
# filter rows
boys <-  filter(boys, !is.na(Rank...1))
```

Finally, we re-arrange the data by stacking them in a three-column format.

```r
# stack columns
boys_long <- bind_rows(boys[,1:3], boys[,4:6])
```

```
## New names:
## * Rank...1 -> Rank
## * Name...2 -> Name
## * Count...3 -> Count

## New names:
## * Rank...7 -> Rank
## * Name...8 -> Name
## * Count...9 -> Count
```

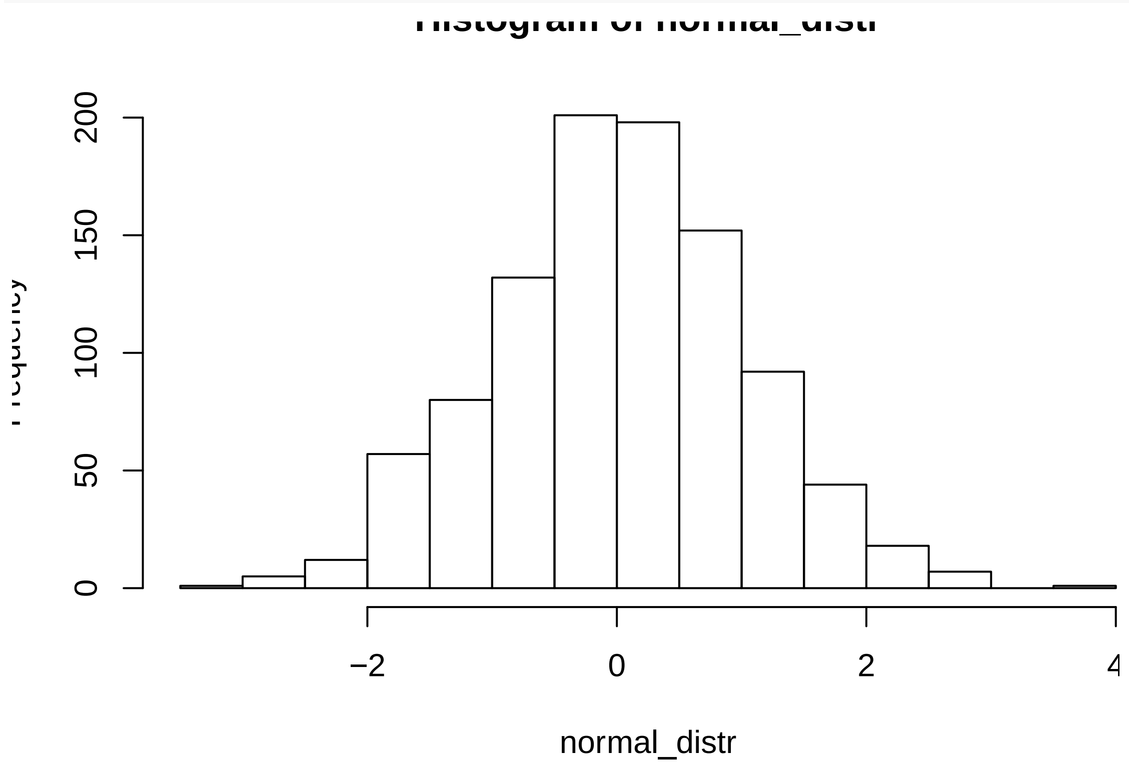
```r
# inspect result
boys_long
```

```
## # A tibble: 114 x 3
##    Rank  Name    Count
##    <chr> <chr>   <dbl>
##  1 1     OLIVER   6941
##  2 2     JACK     5371
##  3 3     HARRY    5308
##  4 4     GEORGE   4869
##  5 5     JACOB    4850
##   6 6     CHARLIE  4831
##   7 7     NOAH     4148
##   8 8     WILLIAM  4083
##   9 9     THOMAS   4075
## 10 10     OSCAR    4066
## # ... with 104 more rows
```

3 Unterstanding statistics and probability with code

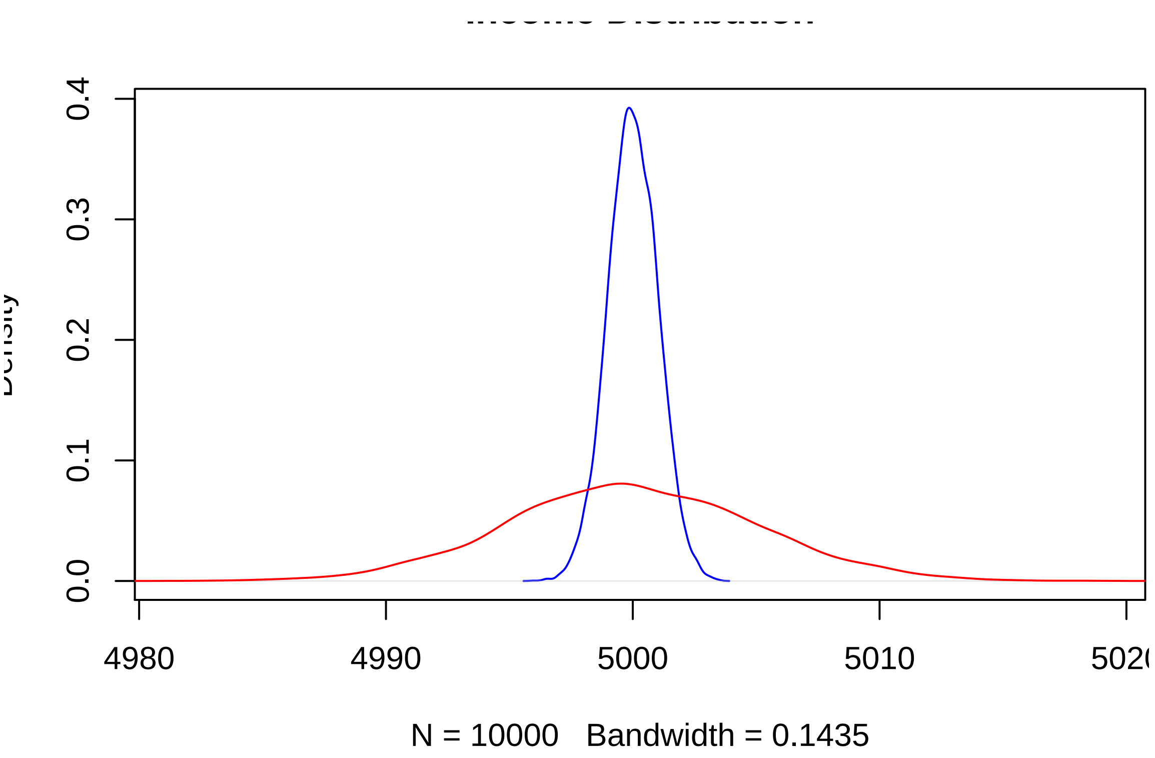
3.1 Random draws and distributions

```
normal_distr <- rnorm(1000)
hist(normal_distr)
```



3.2 Illustration of variability

```
# draw a random sample from a normal distribution with a large standard deviation
largevar <- rnorm(10000, mean = 5000, sd = 5)
# draw a random sample from a normal distribution with a small standard deviation
littlevar <- rnorm(10000, mean = 5000, sd = 1)

# visualize the distributions of both samples with a density plot
plot(density(littlevar), col = "blue",
     xlim=c(min(largevar), max(largevar)), main="Income Distribution")
lines(density(largevar), col = "red")
```



Note: the red curve illustrates the distribution of the sample with a large standard deviation (a lot of variability) whereas the blue curve illustrates the one with a rather small standard deviation.

3.3 Skewness and Kurtosis

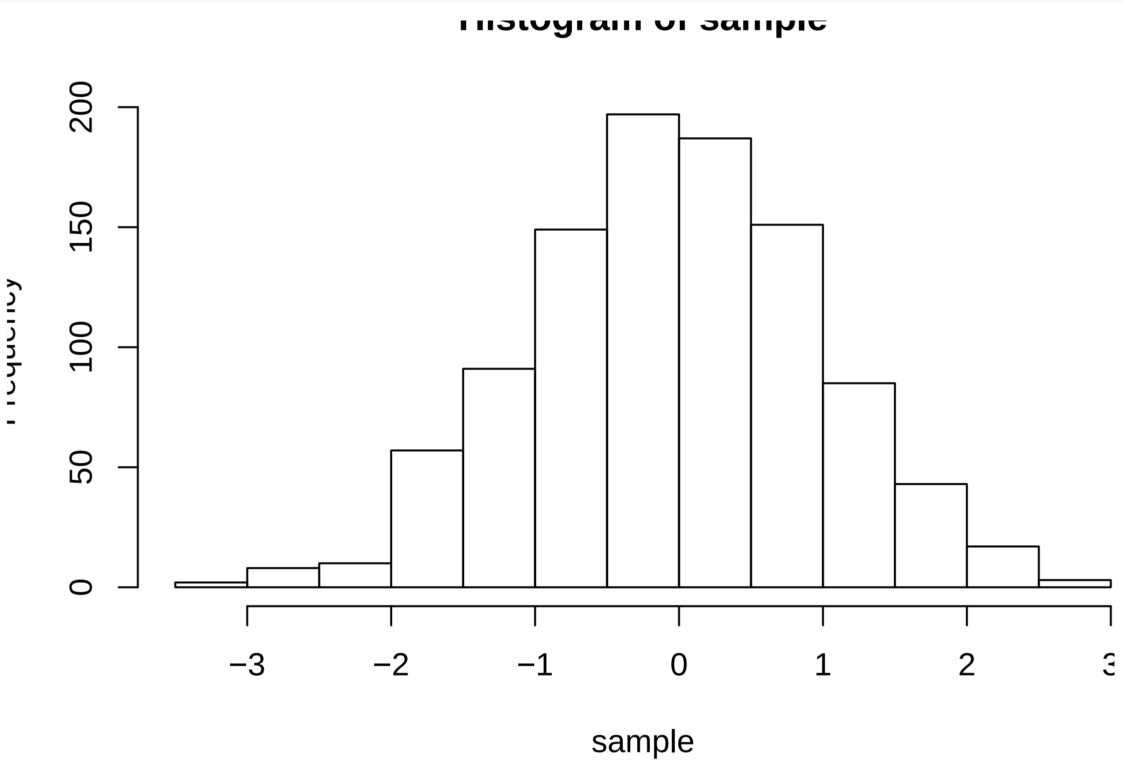
```r
# Install the R-package called "moments" with the following command (if not installed yet):
# install.packages("moments")

# load the package
library(moments)
```

Recall Day 1's slides on Skewness and Kurtosis. A helpful way to memorize what a certain value of either of these two statistics means is to visualize the respective distribution (as shown in the slides).

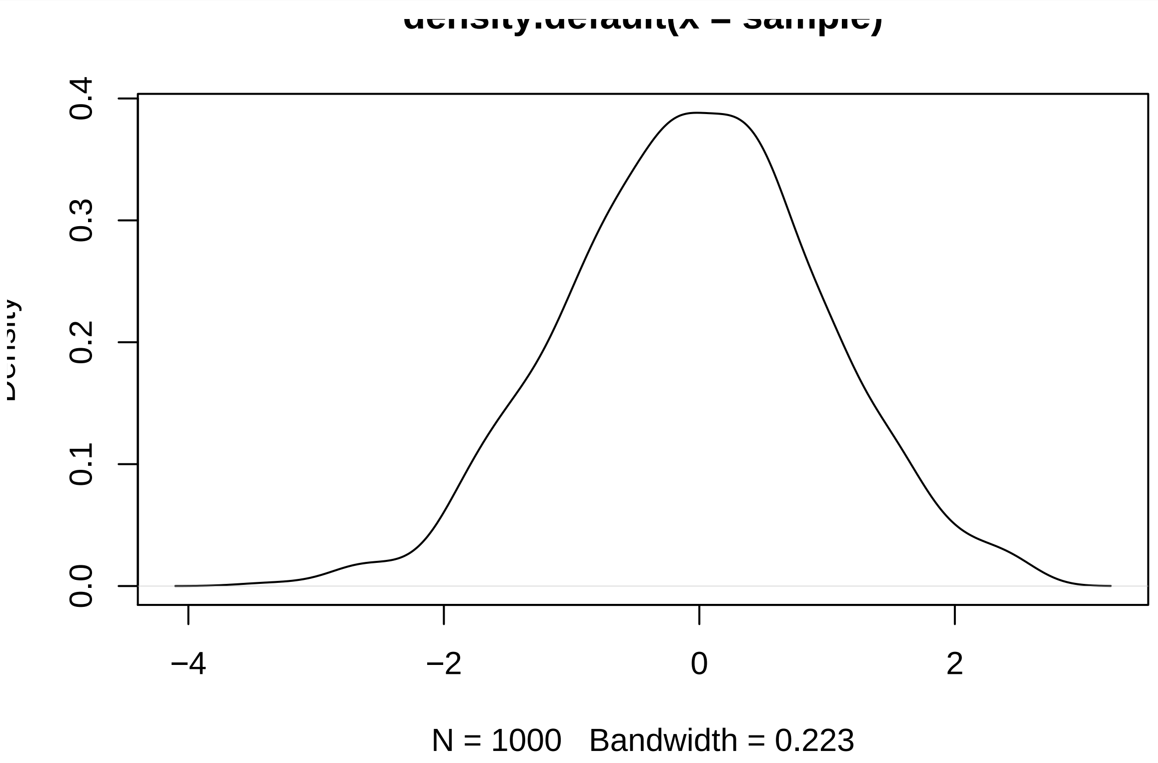
3.3.1 Skewness

Skewness refers to how symmetric the frequency distribution of a variable is. For example, a distribution can be 'positively skewed' meaning it has a long tail on the right and a lot of 'mass' (observations) on the left. We can see that when visualizing the distribution in a histogram or a density plot. In R this looks as follow (consider the comments in the code explaining what each line does):

```r
# draw a random sample of simulated data from a normal distribution
# the sample is of size 1000 (hence, n = 1000)
sample <- rnorm(n = 1000)

# plot a histogram and a density plot of that sample
# note that the distribution is neither strongly positively nor negatively skewed
# (this is to be expected, as we have drawn a sample from a normal distribution)
hist(sample)
```



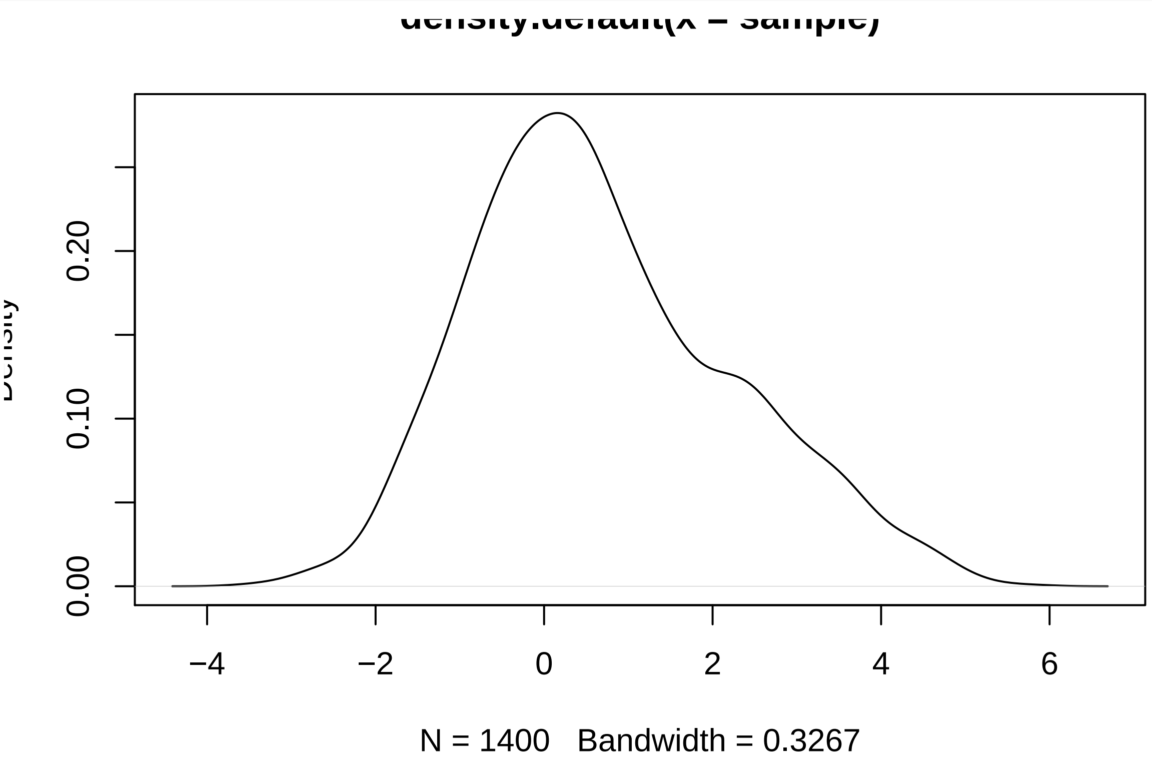
```r
plot(density(sample))
```



```r
# now compute the skewness
skewness(sample)
```

```
## [1] -0.07252817
```

```
# Now we intentionally change our sample to be strongly positively skewed
# We do that by adding some outliers (observations with very high values) to the sample
sample <- c(sample, (rnorm(200) + 2), (rnorm(200) + 3))

# Have a look at the distribution and re-calculate the skewness
plot(density(sample))
```



```
skewness(sample)
```

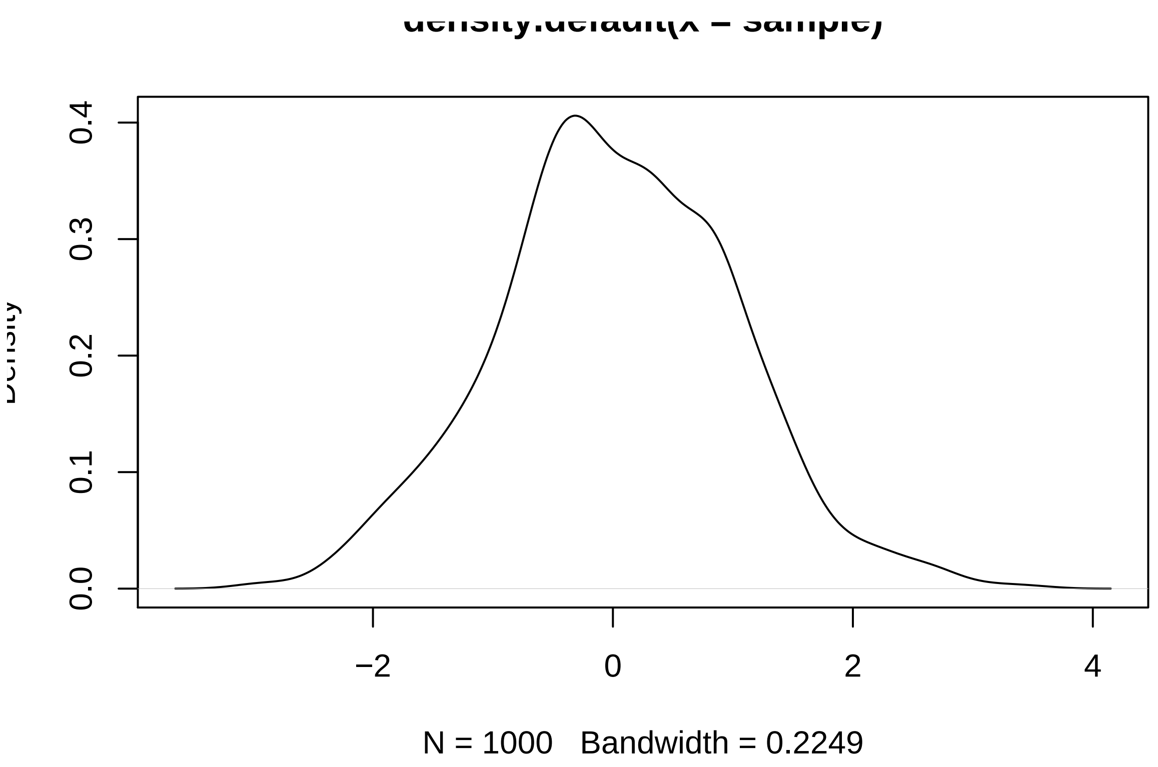
```
## [1] 0.4724813
```

```
#
```

3.3.2 Kurtosis

Kurtosis refers to how much 'mass' a distribution has in its 'tails'. It thus tells us something about whether a distribution tends to have a lot of outliers. Again, plotting the data can help us understand this concept of kurtosis. Lets have a look at this in R (consider the comments in the code explaining what each line does):

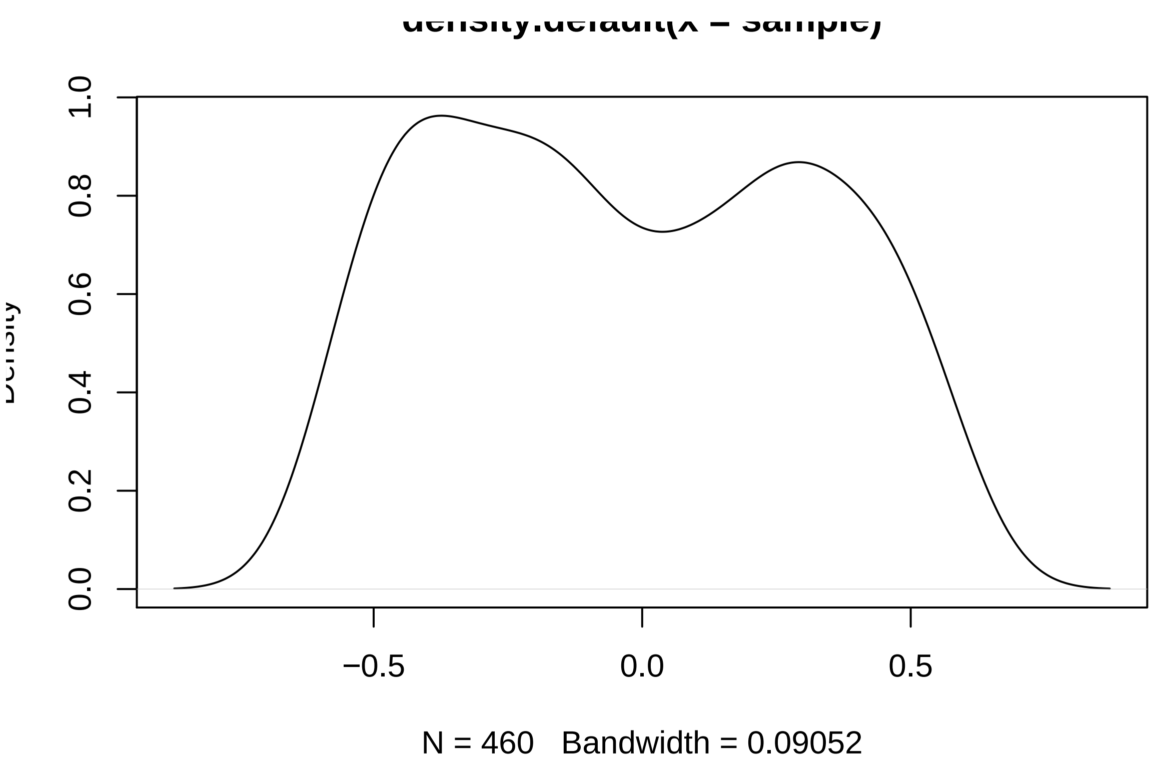
```
# draw a random sample of simulated data from a normal distribution
# the sample is of size 1000 (hence, n = 1000)
sample <- rnorm(n = 1000)

# plot the density & compute the kurtosis
plot(density(sample))
```



```
kurtosis(sample)
```

```
## [1] 3.108739
```

```
# now lets remove observations from the extremes in this distribution
# we thus intentionally alter the distribution to have less mass in its tails
sample <- sample[ sample > -0.6 & sample < 0.6]

# plot the distribution again and see how the tails of it (and thus the kurtosis) has changed
plot(density(sample))
```



```r
# re-calculate the kurtosis
kurtosis(sample)
```

```
## [1] 1.762148
```

```r
# as expected, the kurtosis has now a lower value
```

3.3.3 Implement the formulas for skewness and kurtosis in R

Skewness

```r
# own implementation
sum((sample-mean(sample))^3) / ((length(sample)-1) * sd(sample)^3)
```

```
## [1] 0.0988498
```

```r
# implementation in moments package
skewness(sample)
```

```
## [1] 0.09895742
```

Kurtosis

```r
# own implementation
sum((sample-mean(sample))^4) / ((length(sample)-1) * sd(sample)^4)
```

```
## [1] 1.758318
```

```r
# implementation in moments package
kurtosis(sample)
```

```
## [1] 1.762148
```

3.4 The Law of Large Numbers

The Law of Large Numbers (LLN) is an important statistical property which essentially describes how the behavior of sample averages is related to sample size. Particularly, it states that the sample mean can come as close as we like to the mean of the population from which it is drawn by simply increasing the sample size. That is, the larger our randomly selected sample from a population, the closer is that sample's mean to the mean of the population.

Think of playing dice. Each time we roll a fair die, the result is either 1, 2, 3, 4, 5, or 6, whereby each possible outcome can occur with the same probability (1/6). In other words we randomly draw die-values. Thus we can expect that the average of the resulting die values is $(1 + 2 + 3 + 4 + 5 + 6)/6 = 3.5$.

We can investigate this empirically: We roll a fair die once and record the result. We roll it again, and again we record the result. We keep rolling the die and recording results until we get 100 recorded results. Intuitively, we would expect to observe each possible die-value about equally often (given that the die is fair) because each time we roll the die, each possible value (1,2,..,6) is equally likely to be the result. And we would thus expect the average of the resulting die values to be around 3.5. However, just by chance it can obviously be the case that one value (say 5) occurs slightly more often than another value (say 1), leading to a sample mean slightly larger than 3.5. In this context, the LLN states that by increasing the number of times we are rolling the die, we will get closer and closer to 3.5. Now, let's implement this experiment in R:

```r
# first we define the potential values a die can take
dvalues <- 1:6 # the : operater generates a regular sequence of numbers (from:to)
dvalues
```

```
## [1] 1 2 3 4 5 6
```

```r
# define the size of the sample n (how often do we roll the die...)
# for a start, we only roll the die ten times
n <- 10
# draw the random sample: 'roll the die n times and record each result'
results <- sample( x = dvalues, size = n, replace = TRUE)
# compute the mean
mean(results)
```

```
## [1] 2.9
```

As you can see we are relatively close to 3.5, but not quite there. So let's roll the die more often and calculate the mean of the resulting values again:

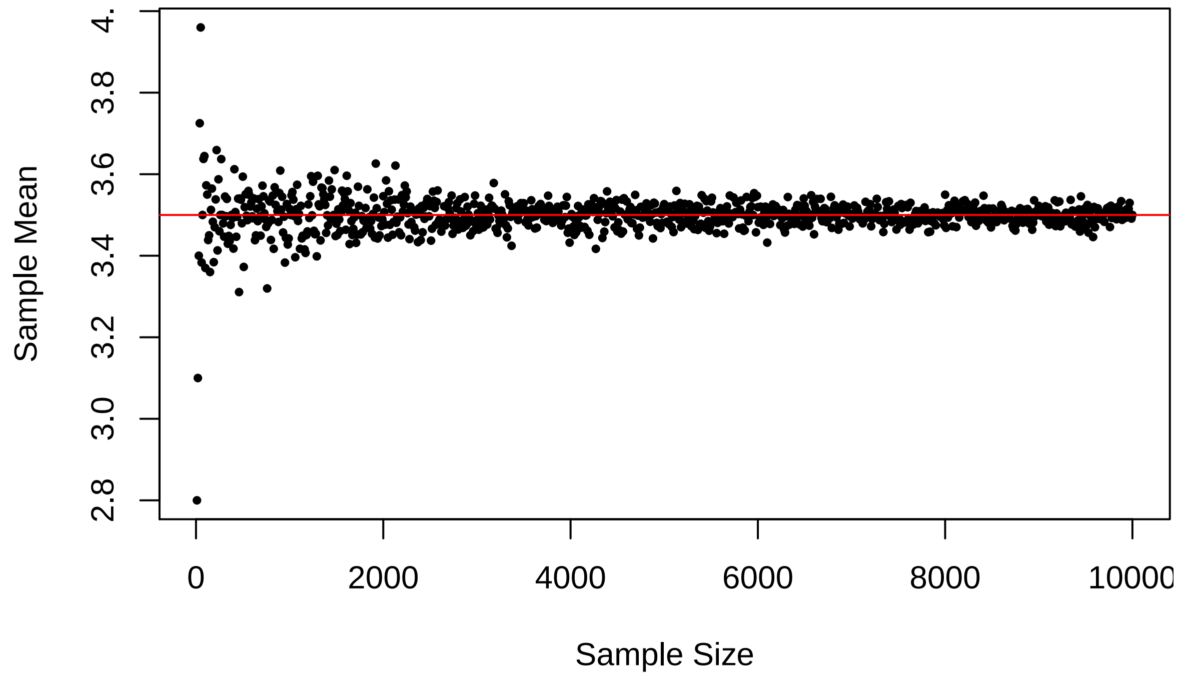
```r
n <- 100
# draw the random sample: 'roll the die n times and record each result'
results <- sample( x = dvalues, size = n, replace = TRUE)
# compute the mean
mean(results)
```

```
## [1] 3.54
```

We are already close to 3.5! Now let's scale up these comparisons and show how the sample means are getting even closer to 3.5 when increasing the number of times we roll the die up to 10'000.

```r
# Essentially, what we are doing here is repeating the experiment above many times,
# each time increasing n.
# define the set of sample sizes
ns <- seq(from = 10, to = 10000, by = 10)
# initiate an empty list to record the results
means <- list()
length(means) <- length(ns)
# iterate through each sample size: 'repeat the die experiment for each sample size'
```

```r
for (i in 1:length(ns)) {

     means[[i]] <- mean(sample( x = dvalues,
                                size = ns[i],
                                replace = TRUE))
}

# visualize the result: plot sample means against sample size
plot(ns, unlist(means),
     ylab = "Sample Mean",
     xlab = "Sample Size",
     pch = 16,
     cex = .6)
abline(h = 3.5, col = "red")
```



We observe that with smaller sample sizes the sample means are broadly spread around the population mean of 3.5. However, the more we go to the right extreme of the x-axis (and thus the larger the sample size), the narrower the sample means are spread around the population mean.

3.5 The Central Limit Theorem

The Central Limit Theorem (CLT) is an almost miraculous statistical property enabling us to test the statistical significance of a statistic such as the mean. In essence, the CLT states that as long as we have a large enough sample, the t-statistic (applied, e.g., to test whether the mean is equal to a particular value) is approximately standard normal distributed. This holds independently of how the underlying data is distributed.

Consider the dice-play-example above. We might want to statistically test whether we are indeed playing with a fair die. In order to test that we would roll the die 100 times and record each resulting value. We would then compute the sample mean and standard deviation in order to assess how likely it was to observe the mean we observe if the population mean actually is 3.5 (thus our H0 would be pop_mean = 3.5, or in plain English 'the die is fair'). However, the distribution of the resulting die values are not standard normal distributed. So how can we interpret the sample standard deviation and the sample mean in the context of our hypothesis?

The simplest way to interpret these measures is by means of a *t-statistic*. A t-statistic for the sample mean under our working hypothesis that pop_mean = 3.5 is constructed as `t(3.5) = (sample_mean - 3.5) / (sample_sd/sqrt(n))`. Let's illustrate this in R:

```r
# First we roll the die like above
n <- 100
# draw the random sample: 'roll the die n times and record each result'
results <- sample( x = dvalues, size = n, replace = TRUE)
# compute the mean
sample_mean <- mean(results)
# compute the sample sd
sample_sd <- sd(results)
# estimated standard error of the mean
mean_se <- sample_sd/sqrt(n)

# compute the t-statistic:
t <- (sample_mean - 3.5) / mean_se
t
```

```
## [1] -0.1780964
```

At this point you might wonder what the use of `t` is if the underlying data is not drawn from a normal distribution. In other words: what is the use of `t` if we cannot interpret it as a value that tells us how likely it is to observe this sample mean, given our null hypothesis? Well, actually we can. And here is where the magic of the CLT comes in: It turns out that there is a mathematical proof (i.e. the CLT) which states that the t-statistic itself can arbitrarily well be approximated by the standard normal distribution. This is true independent of the distribution of the underlying data in our sample! That is, if we have a large enough sample, we can simply compute the t-statistic and look up how likely it is to observe a value at least as large as t, given the null hypothesis is true (-> the p-value):

```r
# calculate the p-value associated with the t-value calculated above
2*pnorm(-abs(t))
```

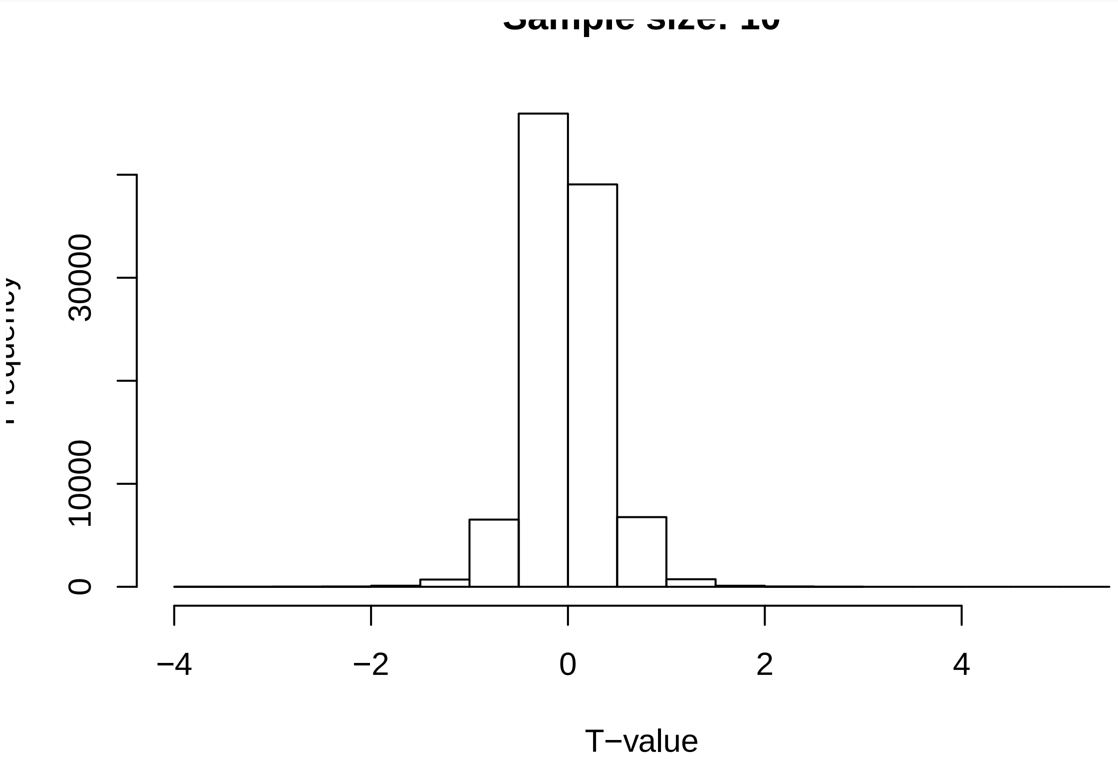
```
## [1] 0.8586472
```

In that case we could not reject the null hypothesis of a fair die. However, as pointed out above, the t-statistic is only asymptotically (meaning with very large samples) normally distributed. We might not want to trust this hypothesis test too much because we were using a sample of only 100 observations.

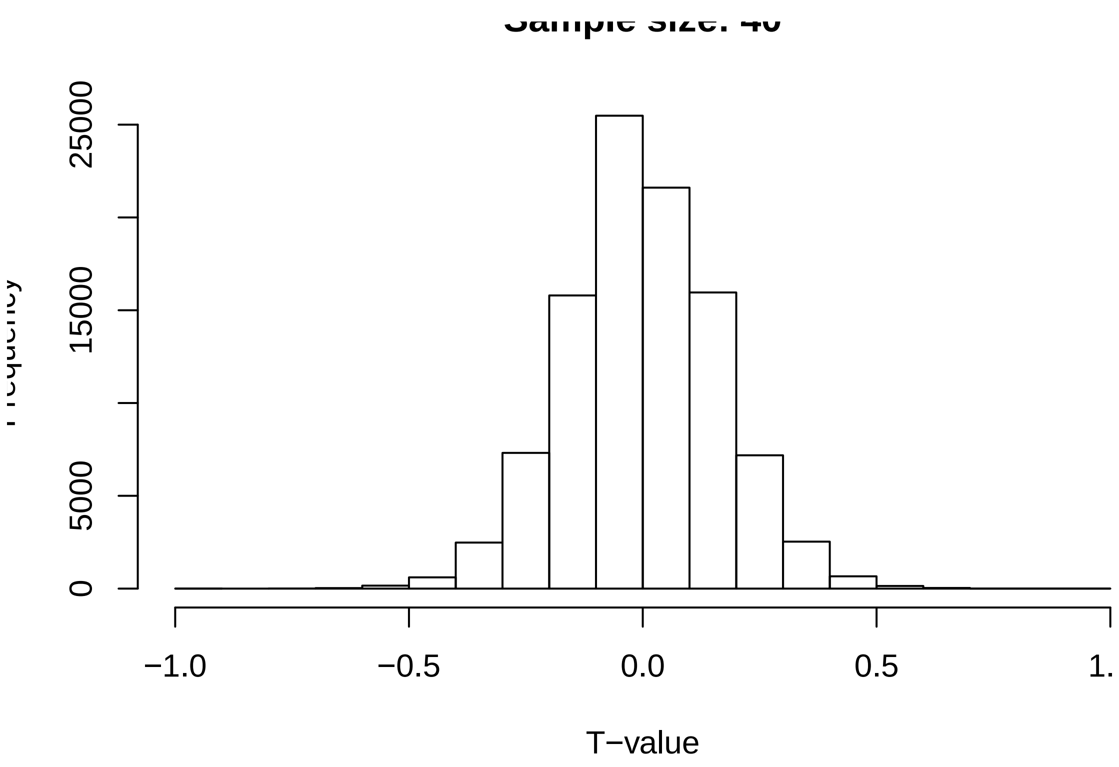
Let's turn back to R in order to illustrate the CLT at work. Similar to the illustration of the LLN above, we will repeatedly compute the t-statistic of our dice play experiment and for each trial increase the number of observations.

```r
# define the set of sample sizes
ns <- c(10, 40, 100)
# initiate an empty list to record the results
ts <- list()
length(ts) <- length(ns)
# iterate through each sample size: 'repeat the die experiment for each sample size'
for (i in 1:length(ns)) {

     samples.i <- sapply(1:100000, function(j) sample( x = dvalues,
                                                      size = ns[i],
                                                      replace = TRUE))
```

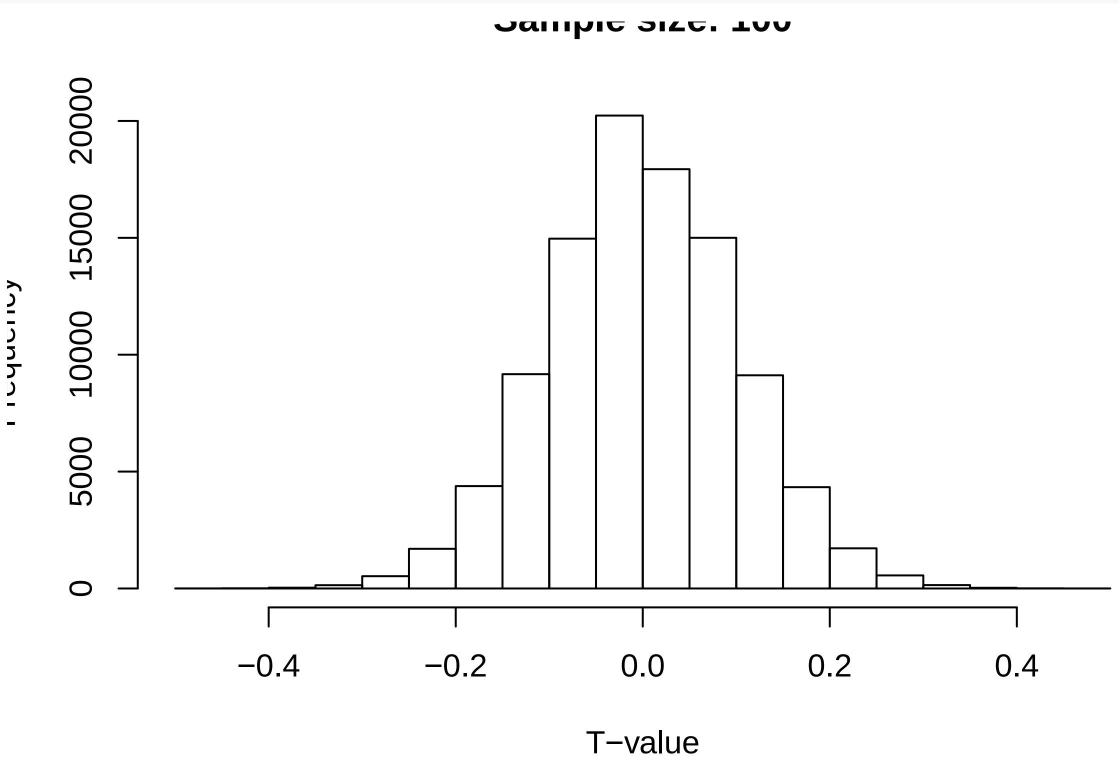
```
    ts[[i]] <- apply(samples.i, function(x) (mean(x) - 3.5) / sd(x), MARGIN = 2)
}

# visualize the result: plot the density for each sample size

# plot the density for each set of t values
hist(ts[[1]], main = "Sample size: 10", xlab = "T-value")
```



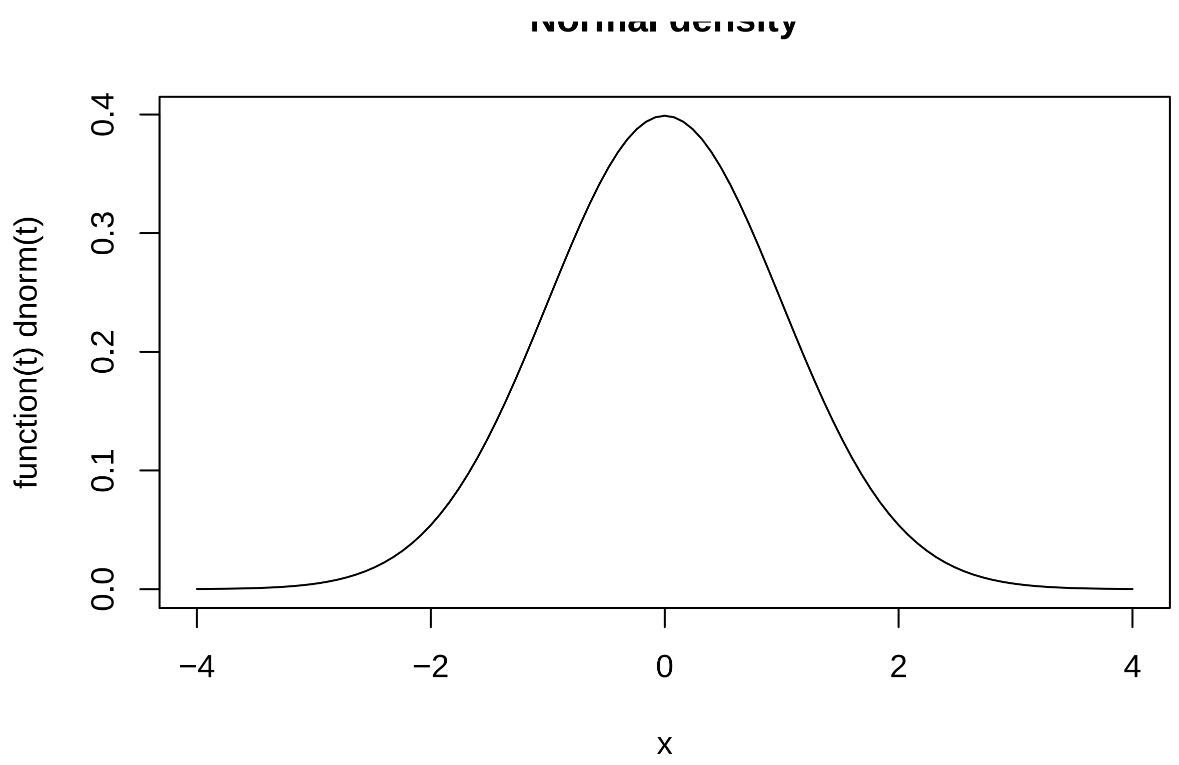
```
hist(ts[[2]], main = "Sample size: 40", xlab = "T-value")
```



```r
hist(ts[[3]], main = "Sample size: 100", xlab = "T-value")
```



```r
# finally have a look at the actual standard normol distribution as a reference point
plot(function(t)dnorm(t), -4, 4, main = "Normal density")
```



Note how the histogram is getting closer to a normal distribution with increasing sample size.