7 The role of simulation in modern statistics

Simulation is at the heart of the computer-intensive methods used in modern statistics. This chapter will introduce simulation and some of its many uses. Particular focus is put on how simulation can be used for analyses and for evaluating the properties of statistical procedures. Most of the content in subsequent chapters can be understood without reading this chapter, but if you're looking for a deeper understanding of modern statistical methods, the topics presented herein are essential.

After reading this chapter, you will be able to use R to:

- Generate random numbers,
- Perform simulations to assess the performance of statistical methods,
- Perform simulation-based hypothesis tests,
- Compute simulation-based confidence intervals, and
- Make sample size computations.

7.1 Simulation and distributions

A *random variable* is a variable whose value describes the outcome of a random phenomenon. A (probability) *distribution* is a mathematical function that describes the probability of different outcomes for a random variable. Random variables and distributions are at the heart of probability theory and most, if not all, statistical models.

As we shall soon see, they are also invaluable tools when evaluating statistical methods. A key component of modern statistical work is *simulation*, in which we generate artificial data that can be used both in the analysis of real data (e.g., in permutation tests and bootstrap confidence intervals, topics that we'll explore in this chapter) and for assessing different methods. Simulation is possible only because we can generate random numbers, so let's begin by having a look at how we can generate random numbers in R.

7.1.1 Generating random numbers

The function `sample` can be used to randomly draw a number of elements from a vector. For instance, we can use it to draw two random numbers from the first 10 integers: $1, 2, \ldots, 9, 10$:

```r
sample(1:10, 2)
```

Try running the above code multiple times. You'll get different results each time, because each time it runs, the random number generator is in a different *state*. In most cases, this is desirable (if the results were the same each time we used `sample`, it wouldn't be random), but not if we want to replicate a result at some later stage.

When we are concerned about reproducibility, we can use `set.seed` to fix the state of the random number generator:

```r
# Each run generates different results:
sample(1:10, 2); sample(1:10, 2)

# To get the same result each time, set the seed to a
# number of your choice:
set.seed(314); sample(1:10, 2)
set.seed(314); sample(1:10, 2)
```

We often want to use simulated data from a probability distribution, such as the normal distribution. The normal distribution is defined by its mean μ and its variance σ^2 (or, equivalently, its standard deviation σ). There are special functions for generating data from different distributions – for the normal distribution it is called `rnorm`. We specify the number of observations that we want to generate (`n`) and the parameters of the distribution (the mean `mu` and the standard deviation `sigma`):

```r
rnorm(n = 10, mu = 2, sigma = 1)

# A shorter version:
rnorm(10, 2, 1)
```

Similarly, there are functions that can be used to compute the quantile function, density function, and cumulative distribution function (CDF) of the normal distribution. Here are some examples for a normal distribution with mean 2 and standard deviation 1:

```
qnorm(0.9, 2, 1)    # Upper 90% quantile of distribution
dnorm(2.5, 2, 1)    # Density function f(2.5)
pnorm(2.5, 2, 1)    # Cumulative distribution function F(2.5)
```

$\sim$

Exercise 7.1 Sampling can be done with or without *replacement*. If replacement is used, an observation can be drawn more than once. Check the documentation for `sample`. How can you change the settings to sample with replacement? Draw five random numbers from the first 10 integers, with replacement.

(Click here to go to the solution.)

7.1.2 Some common distributions

Next, we provide the syntax for random number generation, quantile functions, density/probability functions, and cumulative distribution functions for some of the most commonly used distributions. This section is mainly intended as a reference for you to look up when you need to use one of these distributions – so there is no need to run all the code chunks below right now.

Normal distribution $N(\mu, \sigma^2)$ with mean μ and variance σ^2:

```
rnorm(n, mu, sigma)    # Generate n random numbers
qnorm(0.95, mu, sigma) # Upper 95 %quantile of distribution
dnorm(x, mu, sigma)    # Density function f(x)
pnorm(x, mu, sigma)    # Cumulative distribution function F(X)
```

Continuous uniform distribution $U(a, b)$ on the interval (a, b), with mean $\frac{a+b}{2}$ and variance $\frac{(b-a)^2}{12}$:

```
runif(n, a, b)    # Generate n random numbers
qunif(0.95, a, b) # Upper 95% quantile of distribution
dunif(x, a, b)    # Density function f(x)
punif(x, a, b)    # Cumulative distribution function F(X)
```

Exponential distribution $Exp(m)$ with mean m and variance m^2:

```
rexp(n, 1/m)     # Generate n random numbers
qexp(0.95, 1/m) # Upper 95% quantile of distribution
dexp(x, 1/m)     # Density function f(x)
pexp(x, 1/m)     # Cumulative distribution function F(X)
```

Gamma distribution $\Gamma(\alpha, \beta)$ with mean $\frac{\alpha}{\beta}$ and variance $\frac{\alpha}{\beta^2}$:

```
rgamma(n, alpha, beta)     # Generate n random numbers
qgamma(0.95, alpha, beta) # Upper 95% quantile of distribution
dgamma(x, alpha, beta)     # Density function f(x)
pgamma(x, alpha, beta)     # Cumulative distribution function F(X)
```

Lognormal distribution $LN(\mu, \sigma^2)$ with mean $\exp(\mu + \sigma^2/2)$ and variance $(\exp(\sigma^2) - 1)\exp(2\mu + \sigma^2)$:

```
rlnorm(n, mu, sigma)     # Generate n random numbers
qlnorm(0.95, mu, sigma) # Upper 95% quantile of distribution
dlnorm(x, mu, sigma)     # Density function f(x)
plnorm(x, mu, sigma)     # Cumulative distribution function F(X)
```

t-distribution $t(\nu)$ with mean 0 (for $\nu > 1$) and variance $\frac{\nu}{\nu-2}$ (for $\nu > 2$):

```
rt(n, nu)     # Generate n random numbers
qt(0.95, nu) # Upper 95% quantile of distribution
dt(x, nu)     # Density function f(x)
pt(x, nu)     # Cumulative distribution function F(X)
```

Chi-squared distribution $\chi^2(k)$ with mean k and variance $2k$:

```
rchisq(n, k)     # Generate n random numbers
qchisq(0.95, k) # Upper 95% quantile of distribution
dchisq(x, k)     # Density function f(x)
pchisq(x, k)     # Cumulative distribution function F(X)
```

F-distribution $F(d_1, d_2)$ with mean $\frac{d_2}{d_2-2}$ (for $d_2 > 2$) and variance $\frac{2d_2^2(d_1+d_2-2)}{d_1(d_2-2)^2(d_2-4)}$ (for $d_2 > 4$):

```
rf(n, d1, d2)    # Generate n random numbers
qf(0.95, d1, d2) # Upper 95% quantile of distribution
df(x, d1, d2)    # Density function f(x)
pf(x, d1, d2)    # Cumulative distribution function F(X)
```

Beta distribution $Beta(\alpha, \beta)$ with mean $\frac{\alpha}{\alpha+\beta}$ and variance $\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$:

```
rbeta(n, alpha, beta)    # Generate n random numbers
qbeta(0.95, alpha, beta) # Upper 95% quantile of distribution
dbeta(x, alpha, beta)    # Density function f(x)
pbeta(x, alpha, beta)    # Cumulative distribution function F(X)
```

Binomial distribution $Bin(n, p)$ with mean np and variance $np(1-p)$:

```
rbinom(n, n, p)    # Generate n random numbers
qbinom(0.95, n, p) # Upper 95% quantile of distribution
dbinom(x, n, p)    # Probability function f(x)
pbinom(x, n, p)    # Cumulative distribution function F(X)
```

Poisson distribution $Po(\lambda)$ with mean λ and variance λ:

```
rpois(n, lambda)    # Generate n random numbers
qpois(0.95, lambda) # Upper 95% quantile of distribution
dpois(x, lambda)    # Probability function f(x)
ppois(x, lambda)    # Cumulative distribution function F(X)
```

Negative binomial distribution $NegBin(r, p)$ with mean $\frac{rp}{1-p}$ and variance $\frac{rp}{(1-p)^2}$:

```
rnbinom(n, r, p)    # Generate n random numbers
qnbinom(0.95, r, p) # Upper 95% quantile of distribution
dnbinom(x, r, p)    # Probability function f(x)
pnbinom(x, r, p)    # Cumulative distribution function F(X)
```

Multivariate normal distribution with mean vector μ and covariance matrix Σ:

```
library(MASS)
mvrnorm(n, mu, Sigma) # Generate n random numbers
```

$\sim$

Exercise 7.2 Use `runif` and (at least) one of `round` , `ceiling` and `floor` to generate observations from a discrete random variable on the integers $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$.

(Click here to go to the solution.)

7.1.3 Assessing distributional assumptions

So how can we know that the functions for generating random observations from distributions work? And when working with real data, how can we know what distribution fits the data? One answer is that we can visually compare the distribution of the generated (or real) data to the target distribution. This can for instance be done by comparing a histogram of the data to the target distribution's density function.

To do so, we must add `aes(y = ..density..))` to the call to `geom_histogram` , which rescales the histogram to have area 1 (just like a density function has). We can then add the density function using `geom_function` :

```
# Generate data from a normal distribution with mean 10 and
# standard deviation 1
generated_data <- data.frame(normal_data = rnorm(1000, 10, 1))

library(ggplot2)
# Compare to histogram:
ggplot(generated_data, aes(x = normal_data)) +
      geom_histogram(colour = "black", aes(y = ..density..)) +
      geom_function(fun = dnorm, colour = "red", size = 2,
                    args = list(mean = mean(generated_data$normal_data),
                                sd = sd(generated_data$normal_data)))
```

Try increasing the number of observations generated. As the number of observations increases, the histogram should start to look more and more like the density function.

We could also add a density estimate for the generated data, to further aid the eye here – we'd expect this to be close to the theoretical density function:

```
# Compare to density estimate:
ggplot(generated_data, aes(x = normal_data)) +
      geom_histogram(colour = "black", aes(y = ..density..)) +
      geom_density(colour = "blue", size = 2) +
      geom_function(fun = dnorm, colour = "red", size = 2,
                args = list(mean = mean(generated_data$normal_data),
                               sd = sd(generated_data$normal_data)))
```

If instead we wished to compare the distribution of the data to a χ^2 distribution, we would change the value of `fun` and `args` in `geom_function` accordingly:

```
# Compare to density estimate:
ggplot(generated_data, aes(x = normal_data)) +
      geom_histogram(colour = "black", aes(y = ..density..)) +
      geom_density(colour = "blue", size = 2) +
      geom_function(fun = dchisq, colour = "red", size = 2,
                  args = list(df = mean(generated_data$normal_data)))
```

Note that the values of `args` have changed. `args` should always be a list containing values for the parameters of the distribution: `mu` and `sigma` for the normal distribution and `df` for the χ^2 distribution (the same as in Section 7.1.2).

Another option is to draw a quantile-quantile plot, or Q-Q plot for short, which compares the theoretical quantiles of a distribution to the empirical quantiles of the data, showing each observation as a point. If the data follows the theorised distribution, then the points should lie more or less along a straight line.

To draw a Q-Q plot for a normal distribution, we use the geoms `geom_qq` and `geom_qq_line` :

```
# Q-Q plot for normality:
ggplot(generated_data, aes(sample = normal_data)) +
        geom_qq() + geom_qq_line()
```

For all other distributions, we must provide the quantile function of the distribution (many of which can be found in Section 7.1.2):

```
# Q-Q plot for the lognormal distribution:
ggplot(generated_data, aes(sample = normal_data)) +
        geom_qq(distribution = qlnorm) +
        geom_qq_line(distribution = qlnorm)
```

Q-Q-plots can be a little difficult to read. There will always be points deviating from the line – in fact, that's expected. So how much must they deviate before we rule out a distributional assumption? Particularly when working with real data, I like to compare the Q-Q-plot of my data to Q-Q-plots of simulated samples from the assumed distribution, to get a feel for what kind of deviations can appear if the distributional assumption holds. Here's an example of how to do this, for the normal distribution:

```r
# Look at solar radiation data for May from the airquality
# dataset:
May <- airquality[airquality$Month == 5,]

# Create a Q-Q-plot for the solar radiation data, and store
# it in a list:
qqplots <- list(ggplot(May, aes(sample = Solar.R)) +
  geom_qq() + geom_qq_line() + ggtitle("Actual data"))

# Compute the sample size n:
n <- sum(!is.na(May$Temp))

# Generate 8 new datasets of size n from a normal distribution.
# Then draw Q-Q-plots for these and store them in the list:
for(i in 2:9)
{
    generated_data <- data.frame(normal_data = rnorm(n, 10, 1))
    qqplots[[i]] <- ggplot(generated_data, aes(sample = normal_data)) +
      geom_qq() + geom_qq_line() + ggtitle("Simulated data")
}

# Plot the resulting Q-Q-plots side-by-side:
library(patchwork)
(qqplots[[1]] + qqplots[[2]] + qqplots[[3]]) /
  (qqplots[[4]] + qqplots[[5]] + qqplots[[6]]) /
  (qqplots[[7]] + qqplots[[8]] + qqplots[[9]])
```

You can run the code several times to get more examples of what Q-Q-plots can look like when the distributional assumption holds. In this case, the tail points in the Q-Q-plot for the solar radiation data deviate from the line more than the tail points in most simulated examples do, and, personally, I'd be reluctant to assume that the data comes from a normal distribution.

∼

Exercise 7.3 Investigate the sleep times in the `msleep` data from the `ggplot2` package. Do they appear to follow a normal distribution? A lognormal distribution?

(Click here to go to the solution.)

Exercise 7.4 Another approach to assessing distributional assumptions for real data is to use formal hypothesis tests. One example is the Shapiro-Wilk test for normality, available in `shapiro.test`. The null hypothesis is that the data comes from a normal distribution, and the alternative is that it doesn't (meaning that a low p-value is supposed to imply non-normality).

1. Apply `shapiro.test` to the sleep times in the `msleep` dataset. According to the Shapiro-Wilk test, is the data normally distributed?
2. Generate 2,000 observations from a $\chi^2(100)$ distribution. Compare the histogram of the generated data to the density function of a normal distribution. Are they similar? What are the results when you apply the Shapiro-Wilk test to the data?

(Click here to go to the solution.)

7.1.4 Monte Carlo integration

In this chapter, we will use simulation to compute p-values and confidence intervals to compare different statistical methods, and to perform sample size computations. Another important use of simulation is in *Monte Carlo integration*, in which random numbers are used for numerical integration. It plays an important role in, for instance, statistical physics, computational biology, computational linguistics, and Bayesian statistics; fields that require the computation of complicated integrals.

To create an example of Monte Carlo integration, let's start by writing a function, `circle`, that defines a quarter-circle on the unit square. We will then plot it using the geom `geom_function`:

```r
circle <- function(x)
{
      return(sqrt(1-x^2))
}

ggplot(data.frame(x = c(0, 1)), aes(x)) +
      geom_function(fun = circle)
```

Let's say that we are interested in computing the area under a quarter-circle. We can highlight the area in our plot using `geom_area`:

```r
ggplot(data.frame(x = seq(0, 1, 1e-4)), aes(x)) +
      geom_area(aes(x = x,
                    y = ifelse(x^2 + circle(x)^2 <= 1, circle(x), 0)),
                    fill = "pink") +
      geom_function(fun = circle)
```

To find the area, we will generate a large number of random points uniformly in the unit square. By the law of large numbers, the proportion of points that end up under the quarter-circle should be close to the area under the quarter-circle[49]. To do this, we generate 10,000 random values for the x and y coordinates of each point using the $U(0,1)$ distribution, that is, using `runif`:

```r
B <- 1e4
unif_points <- data.frame(x = runif(B), y = runif(B))
```

Next, we add the points to our plot:

```r
ggplot(unif_points, aes(x, y)) +
      geom_area(aes(x = x,
                    y = ifelse(x^2 + circle(x)^2 <= 1, circle(x), 0)),
                    fill = "pink") +
      geom_point(size = 0.5, alpha = 0.25,
                 colour = ifelse(unif_points$x^2 + unif_points$y^2 <= 1,
                                 "red", "black")) +
      geom_function(fun = circle)
```

Note the order in which we placed the geoms – we plot the points after the area so that the pink colour won't cover the points, and the function after the points so that the points won't cover the curve.

To estimate the area, we compute the proportion of points that are below the curve:

```r
mean(unif_points$x^2 + unif_points$y^2 <= 1)
```

In this case, we can also compute the area exactly: $\int_0^1 \sqrt{1-x^2}dx = \pi/4 = 0.7853\ldots$. For more complicated integrals, however, numerical integration methods like Monte Carlo integration may be required. That being said, there are better numerical integration methods for low-dimensional integrals like this one. Monte Carlo integration is primarily used for higher-dimensional integrals, where other techniques fail.

7.2 Evaluating statistical methods using simulation

An important use of simulation is in the evaluation of statistical methods. In this section, we will see how simulation can be used to compare the performance of two estimators, as well as the type I error rate and power of hypothesis tests.

7.2.1 Comparing estimators

Let's say that we want to estimate the mean μ of a normal distribution. We could come up with several different estimators for μ:

- The sample mean $\bar{x}$,
- The sample median $\tilde{x}$,
- The average of the largest and smallest value in the sample: $\frac{x_{max}+x_{min}}{2}$.

In this particular case (under normality), statistical theory tells us that the sample mean is the best estimator[50]. But how much better is it, really? And what if we didn't know statistical theory? Could we use simulation to find out which estimator to use?

To begin with, let's write a function that computes the estimate $\frac{x_{max}+x_{min}}{2}$:

```
max_min_avg <- function(x)
{
      return((max(x)+min(x))/2)
}
```

Next, we'll generate some data from a $N(0,1)$ distribution and compute the three estimates:

```
x <- rnorm(25)

x_mean <- mean(x)
x_median <- median(x)
x_mma <- max_min_avg(x)
x_mean; x_median; x_mma
```

As you can see, the estimates given by the different approaches differ, so clearly the choice of estimator matters. We can't determine which to use based on a single sample though. Instead, we typically compare the long-run properties of estimators, such as their *bias* and *variance*. The bias is the difference between the mean of the estimator and the parameter it seeks to estimate. An estimator is *unbiased* if its bias is 0, which is considered desirable at least in this setting. Among unbiased estimators, we prefer the one that has the smallest variance. So how can we use simulation to compute the bias and variance of estimators?

The key to using simulation here is to realise that `x_mean` is an observation of the random variable $\bar{X} = \frac{1}{25}(X_1 + X_2 + \cdots + X_{25})$ where each X_i is $N(0,1)$-distributed. We can generate observations of X_i (using `rnorm`), and can therefore also generate observations of $\bar{X}$. That means that we can obtain an arbitrarily large sample of observations of $\bar{X}$, which we can use to estimate its mean and variance. Here is an example:

```
# Set the parameters for the normal distribution:
mu <- 0
sigma <- 1

# We will generate 10,000 observations of the estimators:
B <- 1e4
res <- data.frame(x_mean = vector("numeric", B),
                  x_median = vector("numeric", B),
                  x_mma = vector("numeric", B))

# Start progress bar:
pbar <- txtProgressBar(min = 0, max = B, style = 3)

for(i in seq_along(res$x_mean))
{
      x <- rnorm(25, mu, sigma)
      res$x_mean[i] <- mean(x)
      res$x_median[i] <- median(x)
      res$x_mma[i] <- max_min_avg(x)

      # Update progress bar
      setTxtProgressBar(pbar, i)
}
close(pbar)

# Compare the estimators:
colMeans(res-mu) # Bias
apply(res, 2, var) # Variances
```

All three estimators appear to be unbiased (even if the simulation results aren't exactly 0, they are very close). The sample mean has the smallest variance (and is therefore preferable!), followed by the median. The $\frac{x_{max}+x_{min}}{2}$ estimator has the worst performance, which is unsurprising, as it ignores all information not contained in the extremes of the dataset.

In Section 7.2.5 we'll discuss how to choose the number of simulated samples to use in your simulations. For now, we'll just note that the estimate of the estimators' biases becomes more stable as the number of simulated samples increases, as can be seen from this plot, which utilises `cumsum` , described in Section 5.3.3:

```
# Compute estimates of the bias of the sample mean for each
# iteration:
res$iterations <- 1:B
res$x_mean_bias <- cumsum(res$x_mean)/1:B - mu

# Plot the results:
library(ggplot2)
ggplot(res, aes(iterations, x_mean_bias)) +
      geom_line() +
      xlab("Number of iterations") +
      ylab("Estimated bias")

# Cut the x-axis to better see the oscillations for smaller
# numbers of iterations:
ggplot(res, aes(iterations, x_mean_bias)) +
      geom_line() +
      xlab("Number of iterations") +
      ylab("Estimated bias") +
      xlim(0, 1000)
```

$\sim$

Exercise 7.5 Repeat the above simulation for different sample sizes n between 10 and 100. Plot the resulting variances as a function of n.

(Click here to go to the solution.)

Exercise 7.6 Repeat the simulation in Exercise 7.5, but with a $t(3)$ distribution instead of the normal distribution. Which estimator is better in this case?

(Click here to go to the solution.)

7.2.2 Type I error rate of hypothesis tests

In the same vein that we just compared estimators, we can also compare hypothesis tests or confidence intervals. Let’s have a look at the former and evaluate how well the old-school two-sample t-test fares compared to a permutation t-test and the Wilcoxon-Mann-Whitney test.

For our first comparison, we will compare the type I error rate of the three tests, i.e., the risk of rejecting the null hypothesis if the null hypothesis is true. Nominally, this is the significance level α, which we set to be 0.05.

We write a function for such a simulation, to which we can pass the sizes `n1` and `n2` of the two samples, as well as a function `distr` to generate data:

```r
# Load package used for permutation t-test:
library(MKinfer)

# Create a function for running the simulation:
simulate_type_I <- function(n1, n2, distr, level = 0.05, B = 999,
                            alternative = "two.sided", ...)
{
      # Create a data frame to store the results in:
      p_values <- data.frame(p_t_test = vector("numeric", B),
                             p_perm_t_test = vector("numeric", B),
                             p_wilcoxon = vector("numeric", B))

      # Start progress bar:
      pbar <- txtProgressBar(min = 0, max = B, style = 3)

      for(i in 1:B)
      {
            # Generate data:
            x <- distr(n1, ...)
            y <- distr(n2, ...)

            # Compute p-values:
            p_values[i, 1] <- t.test(x, y,
                                  alternative = alternative)$p.value
            p_values[i, 2] <- perm.t.test(x, y,
                                  alternative = alternative,
                                  R = 999)$perm.p.value
            p_values[i, 3] <- wilcox.test(x, y,
                                  alternative = alternative)$p.value

            # Update progress bar:
            setTxtProgressBar(pbar, i)
      }

      close(pbar)

      # Return the type I error rates:
      return(colMeans(p_values < level))
}
```

First, let's try it with normal data. The simulation takes a little while to run, primarily because of the permutation t-test, so you may want to take a short break while you wait.

```
simulate_type_I(20, 20, rnorm, B = 9999)
```

Next, let's try it with a lognormal distribution, both with balanced and imbalanced sample sizes. Increasing the parameter σ (`sdlog`) increases the skewness of the lognormal distribution (i.e., makes it *more* asymmetric and therefore less similar to the normal distribution), so let's try that too. In case you are in a rush, the results from my run of this code block can be found below it.

```
simulate_type_I(20, 20, rlnorm, B = 9999, sdlog = 1)
simulate_type_I(20, 20, rlnorm, B = 9999, sdlog = 3)
simulate_type_I(20, 30, rlnorm, B = 9999, sdlog = 1)
simulate_type_I(20, 30, rlnorm, B = 9999, sdlog = 3)
```

My results were:

```
# Normal distribution, n1 = n2 = 20:
     p_t_test p_perm_t_test    p_wilcoxon
   0.04760476    0.04780478    0.04680468

# Lognormal distribution, n1 = n2 = 20, sigma = 1:
     p_t_test p_perm_t_test    p_wilcoxon
   0.03320332    0.04620462    0.04910491

# Lognormal distribution, n1 = n2 = 20, sigma = 3:
     p_t_test p_perm_t_test    p_wilcoxon
   0.00830083    0.05240524    0.04590459

# Lognormal distribution, n1 = 20, n2 = 30, sigma = 1:
     p_t_test p_perm_t_test    p_wilcoxon
   0.04080408    0.04970497    0.05300530

# Lognormal distribution, n1 = 20, n2 = 30, sigma = 3:
     p_t_test p_perm_t_test    p_wilcoxon
   0.01180118    0.04850485    0.05240524
```

What's noticeable here is that the permutation t-test and the Wilcoxon-Mann-Whitney test have type I error rates that are close to the nominal 0.05 in all five scenarios, whereas the t-test has too low a type I error rate when the data comes from a lognormal distribution. This makes the test too conservative in this setting. Next, let's compare the power of the tests.

7.2.3 Power of hypothesis tests

The power of a test is the probability of rejecting the null hypothesis if it is false. To estimate that, we need to generate data under the alternative hypothesis. For two-sample tests of the mean, the code is similar to what we used for the type I error simulation above, but we now need two functions for generating data – one for each group, because the groups differ under the alternative hypothesis. Bear in mind that the alternative hypothesis for the two-sample test is that the two distributions differ in location, so the two functions for generating data should reflect that.

```r
# Load package used for permutation t-test:
library(MKinfer)

# Create a function for running the simulation:
simulate_power <- function(n1, n2, distr1, distr2, level = 0.05,
                           B = 999, alternative = "two.sided")
{
      # Create a data frame to store the results in:
      p_values <- data.frame(p_t_test = vector("numeric", B),
                             p_perm_t_test = vector("numeric", B),
                             p_wilcoxon = vector("numeric", B))

      # Start progress bar:
      pbar <- txtProgressBar(min = 0, max = B, style = 3)

      for(i in 1:B)
      {
            # Generate data:
            x <- distr1(n1)
            y <- distr2(n2)

            # Compute p-values:
            p_values[i, 1] <- t.test(x, y,
                                     alternative = alternative)$p.value
            p_values[i, 2] <- perm.t.test(x, y,
                                     alternative = alternative,
                                     R = 999)$perm.p.value
            p_values[i, 3] <- wilcox.test(x, y,
                                     alternative = alternative)$p.value

            # Update progress bar:
            setTxtProgressBar(pbar, i)
      }

      close(pbar)

      # Return power:
```

```r
      return(colMeans(p_values < level))
}
```

Let’s try this out with lognormal data, where the difference in the log means is 1:

```r
# Balanced sample sizes:
simulate_power(20, 20, function(n) { rlnorm(n,
                                      meanlog = 2, sdlog = 1) },
                        function(n) { rlnorm(n,
                                      meanlog = 1, sdlog = 1) },
                        B = 9999)

# Imbalanced sample sizes:
simulate_power(20, 30, function(n) { rlnorm(n,
                                      meanlog = 2, sdlog = 1) },
                        function(n) { rlnorm(n,
                                      meanlog = 1, sdlog = 1) },
                        B = 9999)
```

Here are the results from my runs:

```r
# Balanced sample sizes:
     p_t_test p_perm_t_test    p_wilcoxon
    0.6708671     0.7596760     0.8508851

# Imbalanced sample sizes:
     p_t_test p_perm_t_test    p_wilcoxon
    0.6915692     0.7747775     0.9041904
```

Among the three, the Wilcoxon-Mann-Whitney test appears to be preferable for lognormal data, as it manages to obtain the correct type I error rate (unlike the old-school t-test) and has the highest power (although we would have to consider more scenarios, including different samples sizes, other differences of means, and different values of σ to say for sure!).

Remember that both our estimates of power and type I error rates are proportions, meaning that we can use binomial confidence intervals to quantify the uncertainty in the estimates from our simulation studies. Let’s do that for the lognormal setting with balanced sample sizes, using the results from my runs. The number of simulated samples were 9,999. For the t-test, the estimated type I error rate was `0.03320332`, which corresponds to $0.03320332 \cdot 9,999 = 332$ "successes". Similarly, there were 6,708 "successes" in the power study. The confidence intervals become:

```
library(MKinfer)
binomCI(332, 9999, conf.level = 0.95, method = "clopper-pearson")
binomCI(6708, 9999, conf.level = 0.95, method = "wilson")
```

$\sim$

Exercise 7.7 Repeat the simulation study of type I error rate and power for the old school t-test, permutation t-test, and the Wilcoxon-Mann-Whitney test with $t(3)$-distributed data. Which test has the best performance? How much lower is the type I error rate of the old-school t-test compared to the permutation t-test in the case of balanced sample sizes?

(Click here to go to the solution.)

7.2.4 Power of some tests of location

The `MKpower` package contains functions for quickly performing power simulations for the old-school t-test and Wilcoxon-Mann-Whitney test in different settings. The arguments `rx` and `ry` are used to pass functions used to generate the random numbers, in line with the `simulate_power` function that we created above.

For the t-test, we can use `sim.power.t.test`:

```
library(MKpower)
sim.power.t.test(nx = 25, rx = rnorm, rx.H0 = rnorm,
                 ny = 25, ry = function(x) { rnorm(x, mean = 0.8) },
                 ry.H0 = rnorm)
```

For the Wilcoxon-Mann-Whitney test, we can use `sim.power.wilcox.test` for power simulations:

```
library(MKpower)
sim.power.wilcox.test(nx = 10, rx = rnorm, rx.H0 = rnorm,
                      ny = 15,
                      ry = function(x) { rnorm(x, mean = 2) },
                      ry.H0 = rnorm)
```

7.2.5 Some advice on simulation studies

There are two decisions that you need to make when performing a simulation study:

- How many *scenarios* to include, i.e., how many different settings for the model parameters to study, and
- How many *iterations* to use, i.e., how many simulated samples to create for each scenario.

The number of scenarios is typically determined by what the purpose of the study is. If you only are looking to compare two tests for a particular sample size and a particular difference in means, then maybe you only need that one scenario. On the other hand, if you want to know which of the two tests is preferable in general, or for different sample sizes, or for different types of distributions, then you need to cover more scenarios. In that case, the number of scenarios may well be determined by how much time you have available or how many you can fit into your report.

As for the number of iterations to run, that also partially comes down to computational power. If each iteration takes a long while to run, it may not be feasible to run tens of thousands of iterations (some advice for speeding up simulations by using parallelisation can be found in Section 12.2). In the best of all possible worlds, you have enough computational power available and can choose the number of iterations freely. In such cases, it is often a good idea to use confidence intervals to quantify the uncertainty in your estimate of power, bias, or whatever it is you are studying. For instance, the power of a test is estimated as the proportion of simulations in which the null hypothesis was rejected. This is a binomial experiment, and a confidence interval for the power can be obtained using the methods described in Section 3.5.1. Moreover, the `ssize.propCI` function described in said section can be used to determine the number of simulations that you need to obtain a confidence interval that is short enough for you to feel that you have a good idea about the actual power of the test.

As an example, if a small pilot simulation indicates that the power is about 0.8 and you want a confidence interval with width 0.01, the number of simulations needed can be computed as follows:

```r
library(MKpower)
ssize.propCI(prop = 0.8, width = 0.01, method = "wilson")
```

In this case, you'd need 24,592 iterations to obtain the desired accuracy.

7.3 Sample size computations using simulation

Using simulation to compare statistical methods is a key tool in methodological statistical research and when assessing new methods. In applied statistics, a use of simulation that is just as important is sample size computations. In this section we'll have a look at how simulations can be useful in determining sample sizes.

7.3.1 Writing your own simulation

Suppose that we want to perform a correlation test and want to know how many observations we need to collect. As in the previous section, we can write a function to compute the power of the test:

```
simulate_power <- function(n, distr, level = 0.05, B = 999, ...)
{
      p_values <- vector("numeric", B)

      # Start progress bar:
      pbar <- txtProgressBar(min = 0, max = B, style = 3)

      for(i in 1:B)
      {
            # Generate bivariate data:
            x <- distr(n)

            # Compute p-values:
            p_values[i] <- cor.test(x[,1], x[,2], ...)$p.value

            # Update progress bar:
            setTxtProgressBar(pbar, i)
      }

      close(pbar)

      return(mean(p_values < level))
}
```

Under the null hypothesis of no correlation, the correlation coefficient is 0. We want to find a sample size that will give us 90% power at the 5% significance level, for different hypothesised correlations. We will generate data from a bivariate normal distribution, because it allows us to easily set the correlation of the generated data. Note that the mean and variance of the marginal normal distributions are nuisance variables, which can be set to 0 and 1, respectively, without loss of generality (because the correlation test is invariant under scaling and shifts in location).

First, let’s try our power simulation function:

```
library(MASS) # Contains mvrnorm function for generating data
rho <- 0.5 # The correlation between the variables
mu <- c(0, 0)
Sigma <- matrix(c(1, rho, rho, 1), 2, 2)

simulate_power(50, function(n) { mvrnorm(n, mu, Sigma) }, B = 999)
```

To find the sample size we need, we will write a new function containing a `while` loop (see Section 6.4.5) that performs the simulation for increasing values of n until the test has achieved the desired power:

```r
library(MASS)

power.cor.test <- function(n_start = 10, rho, n_incr = 5, power = 0.9,
                           B = 999, ...)
{
    # Set parameters for the multivariate normal distribution:
    mu <- c(0, 0)
    Sigma <- matrix(c(1, rho, rho, 1), 2, 2)

    # Set initial values
    n <- n_start
    power_cor <- 0

    # Check power for different sample sizes:
    while(power_cor < power)
    {
        power_cor <- simulate_power(n,
                                    function(n) { mvrnorm(n, mu, Sigma) },
                                    B = B, ...)
        cat("n =", n, " - Power:", power_cor, "\n")
        n <- n + n_incr
    }

    # Return the result:
    cat("\nWhen n =", n, "the power is", round(power_cor, 2), "\n")
    return(n)
}
```

Let's try it out with different settings:

```r
power.cor.test(n_start = 10, rho = 0.5, power = 0.9)
power.cor.test(n_start = 10, rho = 0.2, power = 0.8)
```

As expected, larger sample sizes are required to detect smaller correlations.

7.3.2 The Wilcoxon-Mann-Whitney test

The `sim.ssize.wilcox.test` in `MKpower` can be used to quickly perform sample size computations for the Wilcoxon-Mann-Whitney test, analogously to how we used `sim.power.wilcox.test` in Section 7.2.4:

```
library(MKpower)
sim.ssize.wilcox.test(rx = rnorm, ry = function(x) rnorm(x, mean = 2),
                      power = 0.8, n.min = 3, n.max = 10,
                      step.size = 1)
```

∼

Exercise 7.8 Modify the functions we used to compute the sample sizes for the Pearson correlation test to instead compute sample sizes for the Spearman correlation tests. For bivariate normal data, are the required sample sizes lower or higher than those of the Pearson correlation test?

(Click here to go to the solution.)

Exercise 7.9 In Section 3.5.1 we had a look at some confidence intervals for proportions, and saw how `ssize.propCI` can be used to compute sample sizes for such intervals using asymptotic approximations. Write a function to compute the exact sample size needed for the Clopper-Pearson interval to achieve a desired expected (average) width. Compare your results to those from the asymptotic approximations. Are the approximations good enough to be useful?

(Click here to go to the solution.)

7.4 Bootstrapping

The bootstrap can be used for many things, most notably for constructing confidence intervals and running hypothesis tests. These tend to perform better than traditional parametric methods, such as the old-school t-test and its associated confidence interval, when the distributional assumptions of the parametric methods aren’t met.

Confidence intervals and hypothesis tests are always based on a *statistic*, i.e., a quantity that we compute from the samples. The statistic could be the sample mean, a proportion, the Pearson correlation coefficient, or something else. In traditional parametric methods, we start by assuming that our data follows some distribution. For different reasons, including mathematical tractability, a common assumption is that the data is normally distributed. Under that assumption, we can then derive the distribution of the statistic that we are interested in analytically, like Gosset did for the t-test. That distribution can then be used to compute confidence intervals and p-values.

When using a bootstrap method, we follow the same steps, but we use the observed data and simulation instead. Rather than making assumptions about the distribution[51], we use the empirical distribution of the data. Instead of analytically deriving a formula that describes the statistic's distribution, we find a good approximation of the distribution of the statistic by using simulation. We can then use that distribution to obtain confidence intervals and p-values, just as in the parametric case.

The simulation step is important. We use a process known as *resampling*, where we repeatedly draw new observations *with replacement* from the original sample. We draw B samples this way, each with the same size n as the original sample. Each randomly drawn sample – called a *bootstrap sample* – will include different observations. Some observations from the original sample may appear more than once in a specific bootstrap sample, and some not at all. For each bootstrap sample, we compute the statistic in which we are interested. This gives us B observations of this statistic, which together form what is called the *bootstrap distribution* of the statistic. I recommend using $B = 9,999$ or greater, but we'll use smaller B in some examples to speed up the computations.

7.4.1 A general approach

The Pearson correlation test is known to be sensitive to deviations from normality. We can construct a more robust version of it using the bootstrap. To illustrate the procedure, we will use the `sleep_total` and `brainwt` variables from the `msleep` data. Here is the result from the traditional parametric Pearson correlation test:

```r
library(ggplot2)
library(magrittr)

msleep %$% cor.test(sleep_total, brainwt, use = "pairwise.complete")
```

To find the bootstrap distribution of the Pearson correlation coefficient, we can use resampling with a `for` loop (Section 6.4.1):

```r
# Extract the data that we are interested in:
mydata <- na.omit(msleep[,c("sleep_total", "brainwt")])

# Resampling using a for loop:
B <- 999 # Number of bootstrap samples
statistic <- vector("numeric", B)
for(i in 1:B)
{
      # Draw row numbers for the bootstrap sample:
      row_numbers <- sample(1:nrow(mydata), nrow(mydata),
                            replace = TRUE)

      # Obtain the bootstrap sample:
      sample <- mydata[row_numbers,]

      # Compute the statistic for the bootstrap sample:
      statistic[i] <- cor(sample[, 1], sample[, 2])
}

# Plot the bootstrap distribution of the statistic:
ggplot(data.frame(statistic), aes(statistic)) +
         geom_histogram(colour = "black")
```

Because this is such a common procedure, there are R packages that let us do resampling without having to write a `for` loop. In the remainder of the section, we will use the `boot` package to draw bootstrap samples. It also contains convenience functions that allows us to get confidence intervals from the bootstrap distribution quickly. Let's install it:

```r
install.packages("boot")
```

The most important function in this package is `boot`, which does the resampling. As input, it takes the original data, the number B of bootstrap samples to draw (called `R` here), and a function that computes the statistic of interest. This function should take the original data (`mydata` in our example above) and the row numbers of the sampled observation for a particular bootstrap sample (`row_numbers` in our example) as input.

For the correlation coefficient, the function that we input can look like this:

```
cor_boot <- function(data, row_numbers, method = "pearson")
{
    # Obtain the bootstrap sample:
    sample <- data[row_numbers,]

    # Compute and return the statistic for the bootstrap sample:
    return(cor(sample[, 1], sample[, 2], method = method))
}
```

To get the bootstrap distribution of the Pearson correlation coefficient for our data, we can now use `boot` as follows:

```
library(boot)

# Base solution:
boot_res <- boot(na.omit(msleep[,c("sleep_total", "brainwt")]),
                 cor_boot,
                 999)
```

Next, we can plot the bootstrap distribution of the statistic computed in `cor_boot`:

```
plot(boot_res)
```

If you prefer, you can of course use a pipeline for the resampling instead:

```
library(boot)
library(dplyr)

msleep |> select(sleep_total, brainwt) |>
      drop_na() |>
      boot(cor_boot, 999) -> boot_res
```

7.4.2 Bootstrap confidence intervals

The next step is to use `boot.ci` to compute bootstrap confidence intervals. This is as simple as running:

```
boot.ci(boot_res)
```

Four intervals are presented: normal, basic, percentile, and BCa. The details concerning how these are computed based on the bootstrap distribution are presented in Section 14.1. It is generally agreed that the percentile and BCa intervals are preferable to the normal and basic intervals; see, e.g., Davison & Hinkley (1997) and Hall (1992); but, which one performs the best varies.

We also receive a warning message:

```
Warning message:
In boot.ci(boot_res) : bootstrap variances needed for studentized
intervals
```

A fifth type of confidence interval, the studentised interval, requires bootstrap estimates of the standard error of the test statistic. These are obtained by running an *inner bootstrap*, i.e., by bootstrapping each bootstrap sample to get estimates of the variance of the test statistic. Let's create a new function that does this, and then compute the bootstrap confidence intervals:

```r
cor_boot_student <- function(data, i, method = "pearson")
{
    sample <- data[i,]

    correlation <- cor(sample[, 1], sample[, 2], method = method)

    inner_boot <- boot(sample, cor_boot, 100)
    variance <- var(inner_boot$t)

    return(c(correlation, variance))
}

library(ggplot2)
library(boot)

boot_res <- boot(na.omit(msleep[,c("sleep_total", "brainwt")]),
                 cor_boot_student,
                 999)

# Show bootstrap distribution:
plot(boot_res)

# Compute confidence intervals - including studentised:
boot.ci(boot_res)
```

While theoretically appealing (Hall, 1992), studentised intervals can be a little erratic in practice. I prefer to use percentile and BCa intervals instead.

For two-sample problems, we need to make sure that the number of observations drawn from each sample is the same as in the original data. The `strata` argument in `boot` is used to achieve this. Let's return to the example studied in Section 3.6, concerning the difference in how long carnivores and herbivores sleep. Let's say that we want a confidence interval for the difference of two means, using the `msleep` data. The simplest approach is to create a Welch-type interval, where we allow the two populations to have different variances. We can then resample from each population separately:

```
# Function that computes the mean for each group:
mean_diff_msleep <- function(data, i)
{
    sample1 <- subset(data[i, 1], data[i, 2] == "carni")
    sample2 <- subset(data[i, 1], data[i, 2] == "herbi")
    return(mean(sample1[[1]]) - mean(sample2[[1]]))
}

library(ggplot2) # Load the data
library(boot)    # Load bootstrap functions

# Create the data set to resample from:
boot_data <- na.omit(subset(msleep,
              vore == "carni" | vore == "herbi")[,c("sleep_total",
                                                    "vore")])
# Do the resampling - we specify that we want resampling from two
# populations by using strata:
boot_res <- boot(boot_data,
                 mean_diff_msleep,
                 999,
                 strata = factor(boot_data$vore))

# Compute confidence intervals:
boot.ci(boot_res, type = c("perc", "bca"))
```

~

Exercise 7.10 Let's continue the example with a confidence interval for the difference in how long carnivores and herbivores sleep. How can you create a confidence interval under the assumption that the two groups have equal variances?

(Click here to go to the solution.)

7.4.3 Bootstrap hypothesis tests

Writing code for bootstrap hypothesis tests can be a little tricky, because the resampling must be done *under the null hypothesis*. The process is greatly simplified by computing p-values using *confidence interval inversion* instead. This approach exploits the equivalence between confidence intervals and hypothesis tests, detailed in Section 14.2. It relies on the fact that:

- The p-value of the test for the parameter θ is the smallest α such that θ is not contained in the corresponding $1-\alpha$ confidence interval.
- For a test for the parameter θ with significance level α, the set of values of θ that aren't rejected by the test (when used as the null hypothesis) is a $1-\alpha$ confidence interval for θ .

Here is an example of how we can use a `while` loop (Section 6.4.5) for confidence interval inversion, in order to test the null hypothesis that the Pearson correlation between sleep time and brain weight is $\rho = -0.2$. It uses the studentised confidence interval that we created in the previous section:

```r
# Compute the studentised confidence interval:
cor_boot_student <- function(data, i, method = "pearson")
{
    sample <- data[i,]

    correlation <- cor(sample[, 1], sample[, 2], method = method)

    inner_boot <- boot(sample, cor_boot, 100)
    variance <- var(inner_boot$t)

    return(c(correlation, variance))
}

library(ggplot2)
library(boot)

boot_res <- boot(na.omit(msleep[,c("sleep_total", "brainwt")]),
                 cor_boot_student,
                 999)

# Now, a hypothesis test:
# The null hypothesis:
rho_null <- -0.2

# Set initial conditions:
in_interval <- TRUE
alpha <- 0

# Find the lowest alpha for which rho_null is in the interval:
while(in_interval)
{
    # Increase alpha a small step:
    alpha <- alpha + 0.001

    # Compute the 1-alpha confidence interval, and extract
    # its bounds:
    interval <- boot.ci(boot_res,
                        conf = 1 - alpha,
                        type = "stud")$student[4:5]

    # Check if the null value for rho is greater than the lower
    # interval bound and smaller than the upper interval bound,
    # i.e. if it is contained in the interval:
    in_interval <- rho_null > interval[1] & rho_null < interval[2]
}
# The loop will finish as soon as it reaches a value of alpha such
# that rho_null is not contained in the interval.

# Print the p-value:
alpha
```

The `boot.pval` package contains a function computing p-values through inversion of bootstrap confidence intervals. We can use it to obtain a bootstrap p-value without having to write a `while` loop. It works more or less analogously to `boot.ci`. The arguments to the `boot.pval` function is the `boot` object (`boot_res`), the type of interval to use (`"stud"`), and the value of the parameter under the null hypothesis (`-0.2`):

```
install.packages("boot.pval")
library(boot.pval)
boot.pval(boot_res, type = "stud", theta_null = -0.2)
```

Confidence interval inversion fails in spectacular ways for certain tests for parameters of discrete distributions (Thulin & Zwanzig, 2017), so be careful if you plan on using this approach with count data.

~

Exercise 7.11 With the data from Exercise 7.10, invert a percentile confidence interval to compute the p-value of the corresponding test of the null hypothesis that there is no difference in means. What are the results?

(Click here to go to the solution.)

7.4.4 The parametric bootstrap

In some cases, we may be willing to make distributional assumptions about our data. We can then use the *parametric bootstrap*, in which the resampling is done not from the original sample, but the theorised distribution (with parameters estimated from the original sample).

Here is an example for the bootstrap correlation test, where we assume a multivariate normal distribution for the data. Note that we no longer include an index as an argument in the function `cor_boot`, because the bootstrap samples won't be drawn directly from the original data:

```
cor_boot <- function(data, method = "pearson")
{
    return(cor(data[, 1], data[, 2], method = method))
}

library(MASS)
generate_data <- function(data, mle)
{
    return(mvrnorm(nrow(data), mle[[1]], mle[[2]]))
}

library(ggplot2)
library(boot)

filtered_data <- na.omit(msleep[,c("sleep_total", "brainwt")])
boot_res <- boot(filtered_data,
                 cor_boot,
                 R = 999,
                 sim = "parametric",
                 ran.gen = generate_data,
                 mle = list(colMeans(filtered_data),
                            cov(filtered_data)))

# Show bootstrap distribution:
plot(boot_res)

# Compute bootstrap percentile confidence interval:
boot.ci(boot_res, type = "perc")
```

The BCa interval implemented in `boot.ci` is not valid for parametric bootstrap samples, so running `boot.ci(boot_res)` without specifying the interval `type` will render an error[52]. Percentile intervals work just fine, though.

49. In general, the proportion of points that fall below the curve will be proportional to the area under the curve *relative* to the area of the sample space. In this case, the sample space is the unit square, which has area 1, meaning that the relative area is the same as the absolute area.↩

50. At least in terms of mean squared error.↩

51. Well, sometimes we make assumptions about the distribution *and* use the bootstrap. This is known as the parametric bootstrap and is discussed in Section 7.4.4.↩

52. If you really need a BCa interval for the parametric bootstrap, you can find the formulas for it in Davison & Hinkley (1997).↩