Resampling

S.Tonini, F. Chiaromonte (special thanks to J. Di Iorio and L. Insolia)

March 2th 2023

Contents

Introduction

Libraries

We are going to use:

- **boot**: *Bootstrap Functions*
- **coin**: *Conditional Inference Procedures in a Permutation Test Framework*
- **ggplot2**: *Create Elegant Data Visualisations Using the Grammar of Graphics*

```
library(tidyverse)  # data manipulation and visualization
library(boot)       # bootstrapping
library(coin)       # permutation tests
library(ggplot2)    # plots
```

Data

Today we will simulate our dataset! :)

Bootstrapping

There are two different ways of obtaining and evaluating bootstrap estimates:

1. non-parametric;
2. parametric;

- *Goal:* we have a set of n observations from which we are able to calculate a statistic of interest θ, but we have no formula to estimate its standard error. The latter may be useful to construct confidence intervals as well (e.g. ordinary 2-tailed 95%).

Non-parametric Bootstrapping

- *Why non-parametric?* We often cannot reasonably assume that our sample is drawn from a known frequency distribution, but we can assume that it adequately reflects the underlying population from which it was drawn.

By hand

We generate a sample from a binomial distribution with parameters $(15, 0.71)$.

```
set.seed(123)
x <- rbinom(n=30,       # sample size
           size=15,     # num. of trials
           prob=0.71)  # prob. of success per trial
x
```

```
##  [1] 12  9 11  9  8 13 11  8 10 11  8 11 10 10 13  8 12 13 11  8  8 10 10  6 10
## [26] 10 11 10 12 12
```

```
n <- length(x)
n
```

```
## [1] 30
```

```
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    6.00    9.00   10.00   10.17   11.00   13.00
```

Let's pretend that we do not know the true underlying distribution.

Our goal is to estimate the 90th percentile, and we rely on **non-parametric bootstrapping**.

```
help(sample)

set.seed(123)

# number of bootstrap samples
```

```r
B <- 2000

# initialize an empty "container" for each of these
bootstrapsample <- matrix(NA,
                          nrow = n,
                          ncol = B)
dim(bootstrapsample)
```

```
## [1]   30 2000
```

```r
bootstrapsample[1:3, 1:3]
```

```
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]   NA   NA   NA
## [3,]   NA   NA   NA
```

```r
for(i in 1:B){
  # assign (column-wise) B draws with replacement
  bootstrapsample[, i] <- sample(x,n,replace=TRUE)
}

summary(bootstrapsample[, 1:3])
```

```
##        V1              V2               V3
##  Min.   : 8.0   Min.   : 8.00   Min.   : 8.0
##  1st Qu.:10.0   1st Qu.: 9.25   1st Qu.: 9.0
##  Median :10.0   Median :10.00   Median :10.0
##  Mean   :10.3   Mean   :10.20   Mean   :10.4
##  3rd Qu.:11.0   3rd Qu.:11.00   3rd Qu.:12.0
##  Max.   :13.0   Max.   :13.00   Max.   :13.0
```

Now we can compute our statistic of interest on each of these 2000 samples – producing B bootstrap values. In our case, the statistic we are interested in is the 90th percentile.

```r
# to apply some function column-wise
help(apply)
# function of interest
help(quantile)

B_values <- apply(bootstrapsample,   # data
                  2,                 # dimension (i.e. column-wise)
                  quantile,          # function to apply
                  probs=0.9)          # input for the function
head(B_values)
```

```
## [1] 12.0 12.0 13.0 12.0 13.0 11.1
```

So we have the following point estimate and its standard error:

```r
results <- data.frame("mean" = mean(B_values),
                      "SD" = sd(B_values))
rownames(results) <- "Manual"

results
```

```
##            mean        SD
## Manual 12.33695 0.5571399
```

Using the boot package

We can automatically perform non-parametric bootstrapping using the **boot** package. The main bootstrapping function is boot() and has the following syntax:

```r
help(boot)
```

1. **data:** The data as a vector, matrix or data frame. If it is a matrix or data frame then each row is considered as one multivariate observation;
2. **statistic:** A function which when applied to data returns a vector containing the statistic(s) of interest. [...] The first argument passed will always be the original data. The second will be a vector of indices, frequencies or weights which define the bootstrap sample;
3. **R:** The number of bootstrap replicates;

Remark: it is mandatory to pass a"user-defined function" in the field **statistic**.

In the case of the 90th percentile, our estimation function is:

```r
# x: vector
# d: set of indexes
# prob: quantile
sampleperc <- function(x, d, prob=0.9) {
  return(quantile(x[d], probs=prob))
}
```

The estimation function (that we wrote on top, and has to be so) comprises data x and a vector of indexes d. This function will be called many times, one for each bootstrap replication. Every time, the data x will be the same, and only the bootstrap sample indexed by d will change.

Once we have written a function like this, here is how we obtain bootstrap estimates of the standard error for the 90th percentile of the distribution:

```r
set.seed(123)

b = boot(x, sampleperc, R=2000)
print(b)
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
```

```
## Call:
## boot(data = x, statistic = sampleperc, R = 2000)
##
##
## Bootstrap Statistics :
##     original  bias    std. error
## t1*     12.1 0.23385   0.5454704
```

```
# notice how close this is to our previous computation
results <- rbind.data.frame(results,
                            data.frame("mean" = b$t0,   # mean(b$t)
                                       "SD" = sd(b$t))) # for some reason you must compute it again
rownames(results)[2] <- "boot"
results
```

```
##            mean        SD
## Manual 12.33695 0.5571399
## boot   12.10000 0.5454704
```

```
# bias calculation
mean(b$t)-b$t0
```

```
##     90%
## 0.23385
```

It is also easy to get a confidence interval (but be careful) using the function **boot.ci** that requires an object of class "boot" (i.e. computed using **boot**).

This function generates (by default) 5 different types of equi-tailed two-sided nonparametric confidence intervals. These are:

- first order normal approximation: uses the standard deviation for calculation of CI. Use when statistic is unbiased. Is normally distributed.
- basic bootstrap interval: uses percentile to calculate upper and lower limit of test statistic. Use when statistic is unbiased and homoscedastic. The bootstrap statistic can be transformed to a standard normal distribution
- studentized bootstrap interval (bootstrap variance needed): resamples the bootstrap sample to find a second-stage bootstrap statistic and use it to calculate the CI. Use when statistic is homoscedastic. The standard error of bootstrap statistic can be estimated by second-stage resampling.
- bootstrap percentile interval: uses quantiles eg 2.5%, 5% etc. to calculate the CI. Use when statistic is unbiased and homoscedastic. The standard error of your bootstrap statistic and sample statistics are the same.
- adjusted bootstrap percentile (BCa) interval: uses percentile limits with bias correction and estimate acceleration coefficient corrects the limit and find the CI. The bootstrap statistic can be transformed to a normal distribution. The normal-transformed statistic has a constant bias.

```
help(boot.ci)
boot.ci(b, conf=0.95)
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 2000 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = b, conf = 0.95)
##
## Intervals :
## Level      Normal              Basic
## 95%   (10.80, 12.94 )   (11.20, 13.10 )
##
## Level     Percentile            BCa
## 95%   (11.1, 13.0 )   (11.1, 13.0 )
## Calculations and Intervals on Original Scale
```

Parametric Bootstrapping

- *Why parametric?* It is very useful when we can reasonably assume that our sample follows a known distribution.

By hand

We have a sample of size $n = 30$ from a binomial distribution with parameters $(N = 15, p = 0.71)$.

```r
set.seed(123)

n = 30
N <- 15
x <- rbinom(n = n, size = N, prob = 0.71)
x
```

```
##  [1] 12  9 11  9  8 13 11  8 10 11  8 11 10 10 13  8 12 13 11  8  8 10 10  6 10
## [26] 10 11 10 12 12
```

Let's assume that we know the underlying distribution, where the actual parameter p is unknown, and we want to estimate the 90th percentile as before.

Based on MLE: $(\hat{p} = \frac{\sum_{i=1}^{n} x_i}{nN})$

```r
p_hat <- mean(x)/N
p_hat
```

```
## [1] 0.6777778
```

We use parametric bootstrap and compute B samples of size n from the known distribution.

```r
B <- 2000       # number of bootstrap samples
tempdata <- rbinom(B*n,
                   size = N,
                   prob = p_hat)

bootstrapsample <- matrix(tempdata, nrow = n, ncol = B)
dim(bootstrapsample)
```

```
## [1]   30 2000
```

```
bootstrapsample[, 1:5]
```

```
##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    7    9   12   10    8
##  [2,]    8   12   10   11   10
##  [3,]    9   11   11   11   11
##  [4,]    9   11    9   12   11
##  [5,]   13    9   11   11   12
##  [6,]   10   10   12    6   11
##  [7,]    9    9    9   12   10
##  [8,]   12    9   12   13   12
##  [9,]   11    9   10   12   10
## [10,]   12   11   10    9   12
## [11,]   12    9   10   10   10
## [12,]   11   10   11    8   11
## [13,]   11    9   10    9   10
## [14,]   11   15    7    9   11
## [15,]   12   10   10   10   11
## [16,]   12   12    8    9   10
## [17,]   12   11    8    9    9
## [18,]   10   10   10    9   12
## [19,]   11   11   11    6   11
## [20,]    8   12   12   11   11
## [21,]   13   11    7   11   10
## [22,]   10    9   11   11   12
## [23,]    9   11   13   14    8
## [24,]   12    9    7   12    9
## [25,]   10   12    9    8    9
## [26,]   12   11   12   12   10
## [27,]   12    6   10   11   11
## [28,]    9    8    7   13   10
## [29,]    8    8   10   11    8
## [30,]   11   12   11    9   10
```

Now we can compute the statistic of interest on each of these 2000 samples – producing B bootstrap values.

In our case, we are interested in the 90th percentile.

```
B_values <- apply(bootstrapsample, 2, quantile, prob=0.9)
head(B_values)
```

```
## [1] 12.0 12.0 12.0 12.1 12.0 11.1
```

```
summary(B_values)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   11.00   12.00   12.10   12.27   13.00   14.00
```

So we have the following estimate and standard error:

```
resultsParam <- data.frame("mean" = mean(B_values),
                           "SD" = sd(B_values))
rownames(resultsParam) <- "Manual_Parametric"
resultsParam
```

```
##                      mean        SD
## Manual_Parametric 12.2717 0.5362862
```

Let's compare it with the **boot** function:

```
set.seed(123)

B = 2000

p.rg <- function(data, mle, N=15) {
    out <- rbinom(length(data),
                  size = N,
                  prob = mle)
    out
}


bBoot <- boot(x, sampleperc, R = B, sim = "parametric",
              ran.gen = p.rg, mle = (mean(x)/N))
bBoot
```

```
##
## PARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = x, statistic = sampleperc, R = B, sim = "parametric",
##     ran.gen = p.rg, mle = (mean(x)/N))
##
##
## Bootstrap Statistics :
##     original   bias    std. error
## t1*     12.1 0.17165   0.5363069
```

```
# notice how close this is to our previous computation
resultsParam <- rbind.data.frame(resultsParam,
                            data.frame("mean" = bBoot$t0,   # mean(b$t)
                                       "SD" = sd(bBoot$t))) # for some reason you must compute it again
rownames(resultsParam)[2] <- "boot_Parametric"
resultsParam
```

```
##                      mean        SD
## Manual_Parametric 12.2717 0.5362862
## boot_Parametric   12.1000 0.5363069
```

Permutation Test

Permutation tests are particularly relevant in experimental studies, where we are often interested in the sharp null hypothesis of no difference between treatment groups.

Let's generate a dataset divided into treatment (1) and control group (0).

```
# they habe a difference in mean equal to 1
set.seed(1)
n <- 100
tr <- rbinom(n, 1, 0.5)
y <- 1 + tr + rnorm(n, 0, 3)
```

Let us compute the difference in mean between the two groups. The difference in means is, as we would expect (since we made it up), about 1:

```
means <- by(y, tr, mean)
diff0 <- diff(means)
diff0
```

```
## [1] 1.341389
```

To obtain a single permutation of the data, we simply resample without replacement and calculate the difference again:

```
s <- sample(tr, length(tr), FALSE)  # shuffle the labels
by(y, s, mean)                      # compute mean in vector y according to class s
```

```
## s: 0
## [1] 0.8112026
## ------------------------------------------------------------
## s: 1
## [1] 2.094659
```

```
diff(by(y, s, mean))                # difference between 2 means
```

```
## [1] 1.283456
```

If we repeat this process a large number of times, we can build our approximate permutation distribution (i.e., the sampling distribution for the mean-difference).

We'll use **replicate** to repeat our permutation process. The result will be a vector of the differences from each permutation (i.e., our distribution):

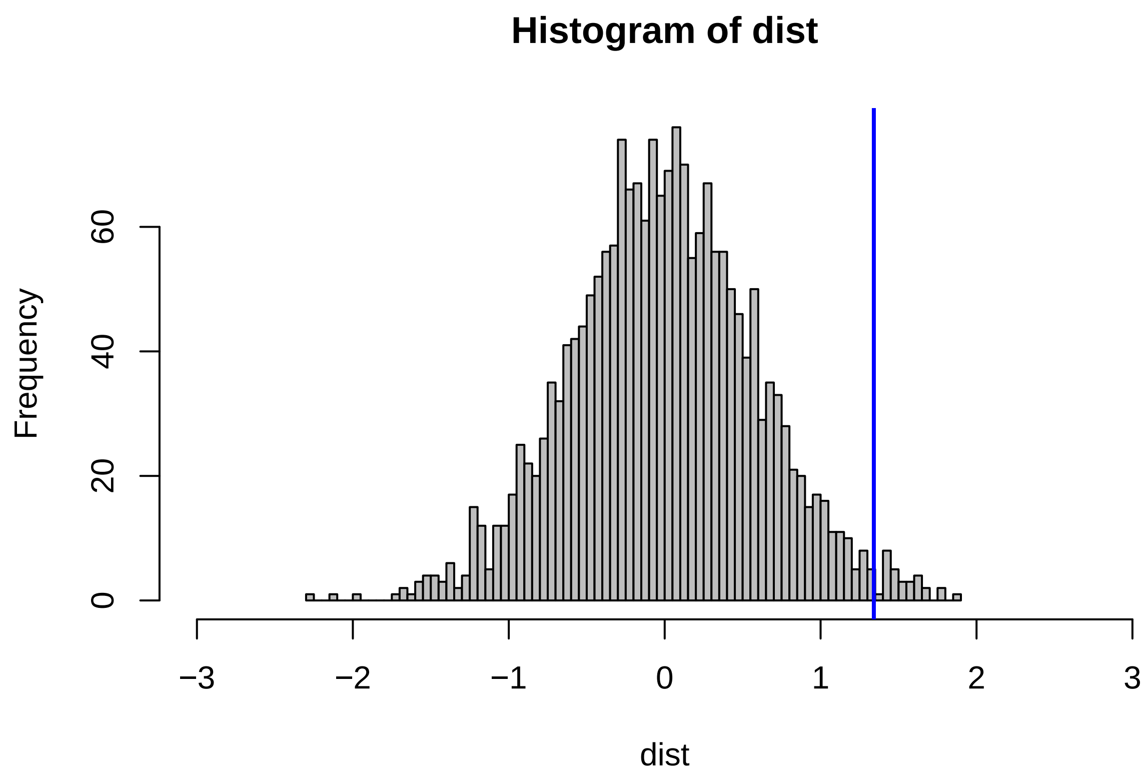
```
set.seed(123)

nperm = 2000
# nperm = 200  # try this too :)
# nperm = 5000
dist <- replicate(nperm, diff(by(y, sample(tr, length(tr), FALSE), mean)))
head(dist)
```

```
##            1            1            1            1            1            1
## -0.906417024 -0.222902177  1.121343866 -0.002785356 -0.116819678  0.289187981
```

We can look at our distribution using an histogram indicating with a vertical line the observed difference:

```
hist(dist, xlim = c(-3, 3), col = "grey", breaks = 100)
abline(v = diff(by(y, tr, mean)), col = "blue", lwd = 2)
```



Now, we can use the distribution to obtain a *p*-value for our mean-difference by counting how many permuted mean-differences are larger than the one we observed in our actual data. We can then divide this by the number of items in our permutation distribution (i.e., nperm=2000 from our call to replicate, above):

```
sum(dist > diff0)/nperm              # one-tailed test
```

```
## [1] 0.0155
```

```
sum(abs(dist) > abs(diff0))/nperm   # two-tailed test
```

```
## [1] 0.029
```

Permutation Test with coin

Even if we implemented our own permutation distributions, R provides a package to conduct permutation tests called **coin**. We can compare our result from above with the result from **coin**:

```r
# library(coin)
independence_test(y ~ tr, alternative = "greater")  # one-tailed
```

```
##
##  Asymptotic General Independence Test
##
## data:  y by tr
## Z = 2.3154, p-value = 0.01029
## alternative hypothesis: greater
```

```r
independence_test(y ~ tr)                           # two-tailed (default)
```

```
##
##  Asymptotic General Independence Test
##
## data:  y by tr
## Z = 2.3154, p-value = 0.02059
## alternative hypothesis: two.sided
```

Almost anything that you can address in a parametric framework can also be done in a permutation framework otherwise you can create your own permutation test!