Bootstrapping Data: Mean

2018-01-09

Preparation

We will use the data in the *iq.csv* file. These data include IQ scores for 20 subjects. The source of these data is: Biostatistics for Dummies.

```
# Load libraries
library(tidyverse)
library(mosaic)
library(sm)

# Read in data
iq_data = read.csv(file = "~/Dropbox/epsy-8252/data/iq.csv")
head(iq_data)
```

iq
61
88
89
89
90
92

```
# Explore IQ scores
iq_data %>%
  summarize(M = mean(iq), SD = sd(iq), N = n())
```

M	SD	N
101	15.8	20

```r
sm.density(iq_data$iq)
```

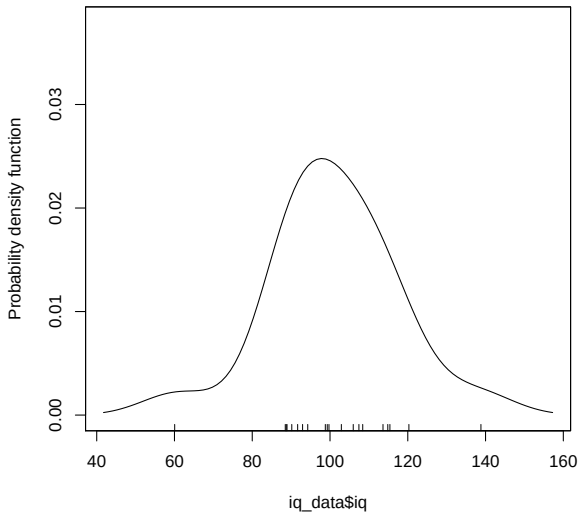

Figure 1. Distribution of 20 IQ scores.

The distribution of IQ scores seems relatively normal ($M = 101$, $SD = 15.8$). One question we might ask is: What is the population mean IQ score?

Our point estimate is 101. But, having taken EPsy 8251, you realize that we should account for uncertainty by estimating the sampling error. We quantify the sampling error in the statistic by computing the *standard error*. In this case, we want to compute the standard error of the mean ($SE_{\bar{X}}$).

Standard Error of the Mean

If we make certain parametric assumptions (assumptions about the population), the standard error of the mean can be computed as:

$$SE_{\bar{X}} = \frac{\sigma}{\sqrt{n}}$$

We can estimate the standard error by using the sample standard deviation as an estimate for σ. In our example,

$$\begin{aligned} \hat{SE}_{\bar{X}} &= \frac{15.8}{\sqrt{20}} \\ &= 3.53 \end{aligned}$$

What does the Standard Error represent?

Recall that the standard error quantifies the amount of uncertainty in the statistic that is due to sampling error. It is the standard deviation (SD) of the *sampling distribution* for that statistic.

Confidence Interval

Another way to report the uncertainty is to compute the 95% confidence interval (CI). The 95% CI for the mean is computed as:

$$\bar{X} \pm 2 \times SE_{\bar{X}}$$

In our case this is,

$$101 \pm 2 \times 3.53,$$

or $[93.9, 108]$. We interpret this as: *With 95% confidence, the population mean IQ score is between 93.9 and 108.* To compute a CI for the mean, we use the `t.test()` function. (This will also output the results of the t-test for testing whether the mean is equal to zero; which we will ignore.)

```
t.test(iq_data$iq)
```

```
##
##  One Sample t-test
##
## data:  iq_data$iq
## t = 30, df = 20, p-value <0.0000000000000002
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##   93.4 108.3
## sample estimates:
## mean of x
##       101
```

This is referred to as the *parametric (normal-theory) confidence interval*, as again, it relies on certain distributional assumptions about the IQ scores (or errors). These assumptions allow us to use the formulaic computation to estimate the standard error.

Bootstrapping the Mean

What if we don't believe the distributional assumption that the errors (or the IQ scores) are normal? Then we have to have a different method for estimating the SE. One empirical method (relying on the sample data) is to estimate the SE via bootstrapping.

Remember the SE is the standard deviation we obtain from the sampling distribution of the mean. The algorithm for this is:

- Draw a SRS size $n = 20$ from the POPULATION.
- Compute the mean for this sample.
- Repeat this process an infinite number of times.
- Compute the SD of all these means. This is the SE.

Bootstrapping uses this same algorithm, but since we don't typically have access to the population in practice, bootstrapping replaces the population in this algorithm with the observed sample. In other words,

- Draw a SRS size $n = 20$ from the OBSERVED SAMPLE. This is called a bootstrap sample.
- Compute the mean for this bootstrap sample (the bootstrapped mean).
- Repeat this process an infinite number of times.
- Compute the SD of all the bootstrapped means. This is our bootstrap estimate of the SE.

There are two complications. First, if we draw a sample of size 20 from the observed sample which is itself size 20, we will get the same sample over and over (no variation in the bootstrapped means; SE = 0). To remedy this issue, we sample WITH REPLACEMENT when we bootstrap.

The second complication is that we can't do this an infinite number of times. We will spend all eternity working on one problem! Instead, in practice, we carry this out a large number of times (1000+). The modified algorithm is then:

- Draw a SRS size $n = 20$ (with replacement) from the OBSERVED SAMPLE. This is called a bootstrap sample.
- Compute the mean for this bootstrap sample (the bootstrapped mean).
- Repeat this process a large number of times (say 1000).
- Compute the SD of all the bootstrapped means. This is our bootstrap estimate of the SE.

Bootstrapping the Mean Using R

To draw a SRS we will use the `sample()` function. We give this function the object we are sampling from (`x=`), the number of elements we want to randomlly sample (`size=`), and whether we are sampling with replacement (`replace=TRUE/FALSE`). Here we randomly sample 20 IQ scores with replacement.

```
sample(x = iq_data$iq, size = 20, replace = TRUE)
```

```
##  [1] 115  93  88  89  90  90 114  89 101  98 101 138 114  88  61 101  61
## [18] 109  90  98
```

To compute the mean of these bootstrapped IQ scores, we wrap that in the `mean()` function.

```
mean(sample(x = iq_data$iq, size = 20, replace = TRUE))
```

```
## [1] 100
```

Now, we need to carry out that computation many, many times. To do this, we will use the `do()` function from the **mosaic** package. This function "does" something K times. It takes the following syntax:

```
do(K) * {function~to~compute}
```

Below we use the `do()` function to bootstrap 20 IQ scores and compute the mean $K = 10$ times.

```
do(10) * {mean(sample(x = iq_data$iq, size = 20, replace = TRUE))}
```

result
101.8
103.0
108.5
97.4
99.2
97.0
97.9
108.3
101.6
98.8

The `do()` function produces a dataframe with a column called `result` that stores the output from our computation. In this case, `result` stores 10 bootstrapped means. In practice, we set K much higher, say 1000, and also assign this into an object so we can compute on it.

```
boot_means = do(1000) * {mean(sample(x = iq_data$iq, size = 20, replace = TRUE))}

# Examine bootstrap distribution
sm.density(boot_means$result)
```

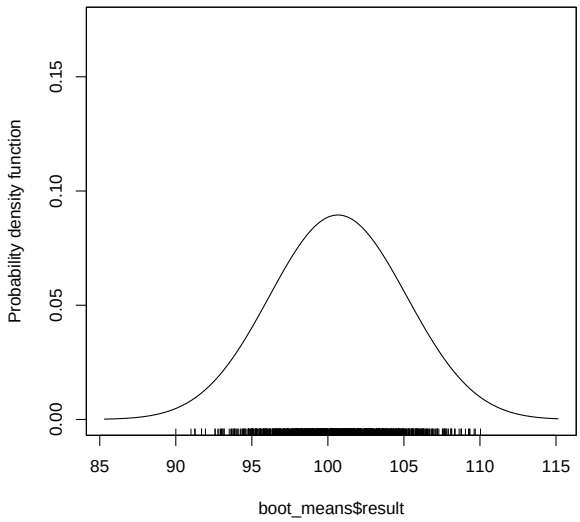

Figure 2. Distribution of 1000 bootstrapped means.

The distribution of bootstrapped means is symmetric

```
# Summary statistics
boot_means %>%
  summarize(M = mean(result), SD = sd(result))
```

M	SD
101	3.46

The bootstrap distribution here has three properties:

- It is symmetric (maybe normal?).
- The mean of the 1000 bootstrapped means is 100.67.
- The SD of the 1000 bootstrapped means is 3.46.

Bootstrap-Based CI: Add/Subtract Two SEs

The estimated SE of the mean from the bootstrap is 3.46. This is what we were originally interested in estimating. We can use this to compute a bootstrap-based 95% CI as:

```
# Lower limit
mean(iq_data$iq) - 2 * sd(boot_means$result)
```

```
## [1] 93.9
```

```
# Upper limit
mean(iq_data$iq) + 2 * sd(boot_means$result)
```

```
## [1] 108
```

In this computation it is important to use the mean of the observed sample. In this case it is the same value as the mean of the bootstrapped means, but depending on how big K is and how skewed the original sample is, they may not be the same value.

Bootstrap-Based CI: Percentile Method

A second way to compute the limits for the confidence interval is to use the *percentile method*. In parametric statistics, the 95% CI represents the limits in the sampling distribution that demarcate the middle 95% of the distribution. In other words it finds the 2.5th- and 97.5th-percentiles in the distribution.

In R, we can use the `quantile()` function to find a percentile. We use it below to find the 2.5th- and 97.5th-percentile of the bootstrapped means.

```
# Lower limit
quantile(boot_means$result, prob = .025)
```

```
## 2.5%
## 93.8
```

```
# Upper limit
quantile(boot_means$result, prob = .975)
```

```
## 97.5%
##   107
```

How does the percentile method of finding the confidence interval compare to the method of adding/subtracting two standard errors?

Table 1.

Lower Limit (LL) and Upper Limit (UL) for Two Bootstrap-Based Methods of Producing Confidence Intervals.

Method	LL	UL
Add/Subtract SE	93.9	108
Percentile Method	93.8	107

These results are nearly identical. This will be true when the bootstrap distribution is normally distributed. In a normal distribution, 95% of the distribution is encompassed within two SDs from the mean—thus both the percentile method and adding/subtracting two SEs will produce roughly the same limits. When the bootstrap distribution is not normally distributed, the percentil method tends to be more correct.

Bootstrapping the Mean: Regression Model

Remember from EPsy 8251 that we can obtain the exact same normal-theory tests and CIs by fitting an appropriate regression model. For example, to obtain the CI for the mean, we fit an intercept-only regression model predicting IQ score and then use the `confint()` function to actually compute the CI.

```
lm.0 = lm(iq ~ 1, data = iq_data)

# Obtain observed mean (marginal mean) IQ score
lm.0
```

```
##
## Call:
## lm(formula = iq ~ 1, data = iq_data)
##
## Coefficients:
## (Intercept)
##         101
```

```
# Obtain CI
confint(lm.0)
```

	2.5 %	97.5 %
(Intercept)	93.4	108

This gives us the exact same CI we obtained using the `t.test()` function. (Furthermore, if we looked at the `summary()` output for the model, we would also get the same p-value for testing whether the mean is equal to zero.)

To bootstrap in a regression model, we need to re-sample/bootstrap *entire rows in the data frame*. With an intercept-only model, this is the same thing as bootstrapping IQ scores. The bootstrap algorithm for regression is then:

- Draw a SRS of $n = 20$ ROWS (with replacement) from the OBSERVED DATAFRAME This is your bootstrap sample.
- Fit the regression model to this bootstrap sample.
- Repeat this process a large number of times (say 1000).
- Compute the SD of all the bootstrapped intercepts. This is our bootstrap estimate of the SE.

We can bootstrap rows from a data frame using the `resample()` function from the **mosaic** package.

```
resample(iq_data)
```

	iq	orig.id
12	102	12
6	92	6
17	114	17
17.1	114	17
11	101	11
1	61	1
14	108	14
11.1	101	11
17.2	114	17
15	109	15
18	115	18
14.1	108	14
16	113	16
14.2	108	14
5	90	5
13	105	13
11.2	101	11
12.1	102	12
3	89	3
14.3	108	14

Since the `resample()` function retains the orignal variable names, we can use this function directly in the `lm()` function—replacing `data=`*dataname* with `data=resample(`*dataname*`)`.

```
lm(iq ~ 1, data = resample(iq_data))
```

```
##
## Call:
## lm(formula = iq ~ 1, data = resample(iq_data))
##
## Coefficients:
## (Intercept)
##        97.3
```

Run this syntax a few times. You will notice that the estimate for β_0 changes as we bootstrap different rows to use in the regression model. It is important to note that the model we are fitting is not different; only the data we are using to fit the model. This means that bootstrapping cannot help you if the initial model you choose is wrong. For example, bootstrapping will not fix a problem where you have fitted a linear model in a situation where the relationship is actually non-linear.

To compute a bootstrap-based CI, we can now use the `do()` function. To illustrate what the `do()` function produces, we will initially set $K = 5$.

```
do(5) * lm(iq ~ 1, data = resample(iq_data))
```

Intercept	sigma	r.squared	.row	.index
96.8	15.93	0	1	1
105.0	13.25	0	1	2
96.2	11.97	0	1	3
100.0	15.53	0	1	4

Intercept	sigma	r.squared	.row	.index
105.3	8.26	0	1	5

By default, when you use `do()` to carry out multiple bootstraps on an `lm()` object, many of the regression estimates you may be interested in are collected for us. In this case, we are interested in the estimates for β_0 (`Intercept`). We will carry out 1000 bootstrap replications and store those replications in an object.

```
boot_reg = do(1000) * lm(iq ~ 1, data = resample(iq_data))
sm.density(boot_reg$Intercept)
```

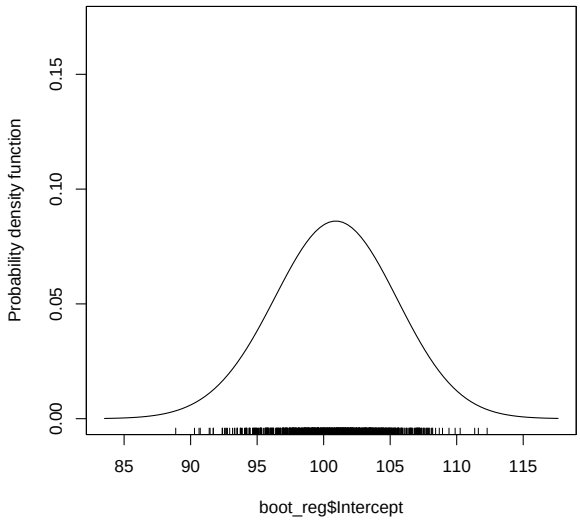

Figure 3. Distribution of 1000 bootstrapped intercepts (means).

The distribution is approximately normal, so either method of obtaining the bootstrap-based confidence interval can be used (they will give essentially the same limits).

```
# Add/subtract SE method
se = sd(boot_reg$Intercept)
mean(iq_data$iq) - 2 * se
```

```
## [1] 93.9
```

```
mean(iq_data$iq) + 2 * se
```

```
## [1] 108
```

```
# Percentile method
quantile(boot_reg$Intercept, prob = c(.025, .975))
```

```
##  2.5% 97.5%
##  93.9 107.4
```

Simulation Error

Bootstrapping is part of a large collection of simulation methods. When we use simulation methods to estimate uncertainty, the uncertainty includes (1) sampling error and (2) simulation error. The goal of estimating the uncertainty for us was to account for sampling error. We need to remove as much of the simulation error as we can to get a good estimate for sampling error.

Simulation error is completely related to the number of replications (the size of K). If we could carry out an infinite number of bootstrap replications, there would be no simulation error. We need to choose a sufficiently large value of K to minimize the amount of simulation error.

How do we do this? Trial and error. Choose successively large values of K. When the estimates do not change from one K to a larger K, you have eliminated much of the simulation error. For example, in estimating the SE, I show below the results of bootstrapping with five different K values.

```
boot_10 = do(10) * lm(iq ~ 1, data = resample(iq_data))
boot_100 = do(100) * lm(iq ~ 1, data = resample(iq_data))
boot_1000 = do(1000) * lm(iq ~ 1, data = resample(iq_data))
boot_10000 = do(10000) * lm(iq ~ 1, data = resample(iq_data))
boot_100000 = do(100000) * lm(iq ~ 1, data = resample(iq_data))
```

Table 2.

Standard Error Estimates and Computational Time (in Seconds) Based on Various Numbers of Bootstrap Replications (K).

K	SE	Time
10	3.09	0.051
100	3.56	0.644
1,000	3.59	5.637
10,000	3.47	47.657
100,000	3.45	507.796

So, if you want the estimate of SE (for these measurements) to the nearest one, it seems that 100 replications are sufficient. To the nearest tenth? Probably also 100 replications. To the nearest hundredth? Maybe 100,000 replications?

So why not just set K to a really high value? Simulation takes computational time. To carry out 100,000 replications took over 8 minutes! And this was using a really simple model with a small dataset. Unfortunately, the `do()` function is not an efficient function for bootstrapping unless $K \leq 5000$.