Simulation, Interval Estimation, and Hypothesis Testing

In this lesson you will see some applications to probability and statistics of what you have learned so far. In particular, you will learn how to estimate the probability of an event, how to find an interval estimate for an unknown parameter, and how to provide evidence for a claim.

Simulation

Ferdinando dei Medici, the Grand Duke of Tuscany, asked Galileo the following question. Which is more likely, a sum of 9, or a sum of 10, when rolling 3 dice? You can answer this question with some basic knowledge of probability theory. There is, however, a more simple minded approach. Just roll three dice many times and count which event occurs more frequently.

This approach would be very time consuming. To speed it up we could simulate the rolls of the dice with software. First, we would need a way to randomly pick one of the six numbers 1, 2, 3, 4, 5, and 6. There is a function for that in R

```
sample(1:6, size=1)
```

How can software which is deterministic do something random? Well, it really can't but it can make it look like it can. If you run the line of code above several times then the results will be for all practical purposes be like those you might obtain by rolling a die several times.

Here is an example of a random number generator. Start with a number x between 0 and 1. Add π, raise the result to the 8-th power and take the fractional part as the next "random number".

```
rand <- function(x){
    y <- (x+pi)^8
    y-floor(y)
}
```

Repeat this process to obtain a sequence of say n=100 "random numbers" between 0 and 1.

```
n <- 100
start <- 0.2341
rand.numbers <- numeric(n)
rand.numbers[1] <- rand(start)
for (i in 2:n){rand.numbers[i] <- rand(rand.numbers[i-1])}
```

We can use these 100 "random numbers" to simulate 100 rolls of a die.

```
ceiling(rand.numbers*6)
```

The random number generators implemented in R are much more sophisticated, but the algorithm described above will give you an idea on how this might work.

Here is how we can simulate the roll of three dice and count the frequencies of nines and tens.

```
galileo=function(n){
    freq9 <- 0 # initialize counting variable
    freq10 <- 0 # initialize counting variable
    for (i in 1:n){
        # roll the three dice and record the sum
        dice <- sum(sample(1:6, size=3, replace=TRUE))
        # if the sum is 9 then increase the counter freq9 by 1
        if (dice == 9) {freq9 <- freq9 + 1}
        # if the sum is 10 then increase the counter freq10 by 1
        if (dice == 10) {freq10 <- freq10 + 1}
    }
    cat('In',n,'rolls of 3 dice the sum was nine', 100*freq9/n, '% and the sum
was ten', 100*freq10/n,'% of the times.')
}
```

Then we can get the result of 10,000 repetitions of rolling three dice as follows.

```
galileo(10000)
```

Anytime you run this function your output will be different. This is not surprising, since the result is based on simulated rolls of three dice. We can however make this result reproducible by using the function *set.seed()*. This is like setting the starting number in the random number generator we discussed above.

```
set.seed(7000) # to make the result reproducible
galileo(10000)
```

Now, every time you run this code the result will be the same and therefore reproducible. Others can now check your answers. As you will see later, the probability for rolling a 9 is 25/216 or 0.1157407 and the probability for rolling a 10 is 27/216 or 0.125.

We expect the results of the simulation to be better if they are based on more repetitions of the random experiment. If we used a billion repetitions then the result should be closer to the exact answer than if we used ten thousand repetitions. How long would it take to run one billion repetitions?

We can time how long the evaluation of a function takes with the function *system.time()*.

```
system.time(galileo(10000))[3]
```

On my old laptop (MacBook Air, 2011) the elapsed time for evaluating this function was 0.149 seconds. Running the same code again will typically give a different answer. The reason is that your computer manages many processes and has to divide its resources between these.

If 10,000 repetitions took 0.149 seconds on my computer, then 1 billion repetitions are estimated to take 100,000 times as long, that is, 100000*0.149 seconds. This is about 248 minutes or about 4 hours.

It is possible that your computer is so fast that the elapsed time for 10,000 repetitions is reported as 0 seconds. In this case increase the number of repetitions to 100,000 and then multiply the elapsed time by 10,000.

I could run 1 billion repetitions on my old laptop if I'd be willing to wait about 4 hours. In such a case it is worth investigating if a faster algorithm can be found.

```
galileo2=function(n){
    x <- matrix(data=sample(1:6, size=3*n, replace=TRUE), nrow=3)
    y <- colSums(x)
    freq9 <- sum(y == 9)
    freq10 <- sum(y == 10)
    cat('In',n,'rolls of 3 dice the sum was nine', 100*freq9/n, '% and the sum
was ten', 100*freq10/n,'% of the times.')
}
system.time(galileo2(1e6))[3]
```

Running 1 million repetitions took 0.354 seconds and therefore running 1 billion repetitions is estimated to take about 6 minutes. That's much better than 4 hours. Unfortunately there is another problem. The matrix x defined in this function has 3 billion entries which would need about 12 Gigabytes of memory (each integer uses 4 bytes). My old laptop has only 4 Gigabytes of RAM and so this will not work for me.

Interval Estimates

Suppose that you have a random sample from some population and you have a method that lets you calculate an estimate $\hat{\theta}$ for an unknown parameter θ of this population.

Instead of a point estimate you would like an interval estimate. This can be accomplished by resampling the sample. There are several methods and I chose the following method for its simplicity (see Rice, *Mathematical Statistics and Data Analysis*, 3rd edition, Duxbury 2007, page 284 for a better motivated approach).

Resample the random sample 1000 times and calculate the estimate $\hat{\theta}$ for each sample. Sort these 1000 estimates and take the 25th and the 975th value as your lower and upper bound of your interval estimate.

This method is an example of a *nonparametric bootstrap*. More specifically it is known as the *bootstrap percentile method.*

To illustrate this method suppose we want to find an interval estimate of the population mean and suppose we collected the following sample.

```
x <- c(0.7, 0.52, 1.88, 1.42, 0.42, 0.4, 1.31, 0.63, 0.1, 0.69, 1.75, 0.95,
0.42, 0.29, 0.44, 1.46, 2.02, 0.06, 6.46, 2.22, 0.28, 0.38, 0.55, 1.55, 0.24,
0.37, 1.88, 0.77, 0.17, 0.77, 0.61, 1.96, 0.08, 1.28, 3.37, 0.01, 2.76, 0.22,
0.8, 1, 0.43, 0.5, 2.63, 0.76, 0.22, 0.38, 0.22, 0.15, 1.14, 0.21, 0.85, 0.1,
0.69, 0.75, 1.35, 0.8, 0.36, 0.66, 0.47, 0.13)
```

We resample this sample 1000 times, calculate the average in each sample, and store the result in the vector *estimates*.

```
estimates <- numeric(1000)
set.seed(7000)
for (i in 1:1000){
    new.sample <- sample(x, size=60, replace=TRUE)
    estimates[i] <- mean(new.sample)
}
```

Now sort these 1000 estimates and take the 25th and the 975th value as our lower and upper bound of an interval estimate.

```
sort(estimates)[c(25, 975)]
```

The interpretation of this interval is that we can be 95% confident that the true mean is contained in this interval.

How could we check if this method really works? We can repeat this method with samples from a population with known mean many times and check how often the interval contains the population mean.

Hypothesis Testing

Suppose we came up with a new approach to a particular problem and we want to provide evidence that our new approach is better. To this end we perform an experiment and collect the following data.

```
# data from standard approach
x <- c(3.9, 1.87, 3.11, 5.1, 4.28, 5.56, 3.76, 4.12, 3.05, 3.75)
# data from new approach
y <- c(4.91, 3.26, 4.55, 2.79, 3.86, 1.89, 5.59, 3.08)
```

Suppose that smaller values are better. The average value from the standard approach is 3.85 and the average value from the new approach is 3.74125. It looks like our new approach is better. However, this difference might be due to the variation in the outcomes of the experiment.

Suppose for a moment that the data for x and the data for y come from the same distribution and it just so happened by chance that the average of our values for the standard approach were smaller. The difference between the averages is $3.85 - 3.74125 = 0.10875$. How likely is it that such a difference occurs by chance alone?

We can estimate this chance as follows. We combine the two data sets and form 10,000 random permutations of the combined sample. We split each random permutation into two groups of measurements where one group represents data from the standard approach and the other represents data from the new approach. We calculate the difference between the two groups as before for each of the 10,000 random permutations. Now we count how often this difference was as large or larger than the difference between the actual groups. Divide this number by 10,000. This number is the relative frequency (or proportion) of times a certain measure of difference between the two data sets was as large or larger than the same measure of difference between the actual data sets. If this number is small then we interpret this as evidence that the new approach produces a smaller average value.

```
z <- c(x, y)
difference <- numeric(10000)
for (i in 1:10000){
    z.shuffled <- sample(z)
    x.new <- z.shuffled[1:10]
    y.new <- z.shuffled[11:18]
    difference[i] <- mean(x.new) - mean(y.new)
}
sum(difference >= 0.10875)/10000
```

In our case the relative frequency is 0.4206. This is not small and so we cannot claim that we have evidence that the new approach is better.