Lab 7: Simulation

Statistical Computing, 36-350

Week of Monday October 8, 2018

Name:
Andrew ID:
Collaborated with:

This lab is to be done in class (completed outside of class if need be). You can collaborate with your classmates, but you must identify their names above, and you must submit **your own** lab as an knitted HTML file on Canvas, by Sunday 11:59pm, this week.

This week's agenda: practice writing functions and running simulations.

Basic random number generation

- **1a.** Generate the following objects, save them to variables (with names of your choosing), and call `head()` on those variables.
  - A vector with 1000 standard normal random variables.
  - A vector with 20 draws from $\mathrm{Beta}(0.1, 0.1)$.
  - A vector of 2000 characters sampled uniformly from "A", "G", "C", and "T".
  - A data frame with a column `x` that contains 100 draws from $\mathrm{Unif}(0, 1)$, and a column `y` that contains 100 draws of the form $y_i \sim \mathrm{Unif}(0, x_i)$. Do this without using explicit iteration.
- **1b.** We've written a function `plot.cum.means()` below which plots cumulative sample mean as the sample size increases. The first argument `rfun` stands for a function which takes one argument `n` and generates this many random numbers when called as `rfun(n)`. The second argument `n.max` is an integer which tells the number samples to draw. As a side effect, the function plots the cumulative mean against the number of samples.

  ```r
  # plot.cum.means: plot cumulative sample mean as a function of sample size
  # Inputs:
  # - rfun: function which generates random draws
  # - n.max: number of samples to draw
  # Ouptut: none
  plot.cum.means = function(rfun, n.max) {
    samples = rfun(n.max)
    plot(1:n.max, cumsum(samples) / 1:n.max, type = "l")
  }
  ```

  Use this function to make plots for the following distributions, with `n.max=1000`. Then answer: do the sample means start concentrating around the appropriate value as the sample size increases?

  - $N(-3, 10)$
  - $\mathrm{Exp}(\mathrm{mean} = 5)$
  - $\mathrm{Beta}(1, 1)$

  Hint: for each, you should construct a new single-argument random number generator to pass as the `rfun` argument to `plot.cum.means()`, as in `function(n) rnorm(n, mean=-3, sd=sqrt(10))` for the first case.
- **Challenge.** Find a distribution whose sample mean should not converge (in theory) as the sample size grows. Call `plot.cum.means()` with the appropriate random number generator and `n.max=1000`.

- **1c.** For the same distributions as Q1b we will do the following.
  - Generate 10, 100, and 1000 random samples from the distribution.
  - On a single plot, display the ECDFs (empirical cumulative distribution functions) from each set of samples, and the true CDF, with each curve being displayed in a different color.

  In order to do this, we'll write a function `plot.ecdf(rfun, pfun, sizes)` which takes as its arguments the single-argument random number generating function `rfun`, the corresponding single-argument conditional density function `pfun`, and a vector of sample sizes `sizes` for which to plot the ecdf.

  We've already started to define `plot.ecdf()` below, but we've left it incomplete. Fill in the definition by editing the lines with "##" and "??", and then run it on the same distributions as in Q1b. Examine the plots and discuss how the ECDFs converge as the sample size increases. Note: make sure to remove `eval=FALSE`, after you've edited the function, to see the results.

```r
# plot.ecdf: plots ECDFs along with the true CDF, for varying sample sizes
# Inputs:
# - rfun: function which generates n random draws, when called as rfun(n)
# - pfun: function which calculates the true CDF at x, when called as pfun(x)
# - sizes: a vector of sample sizes
# Output: none

plot.ecdf = function(rfun, pfun, sizes) {
  # Draw the random numbers
  ## samples = lapply(sizes, ??)

  # Calculate the grid for the CDF
  grid.min = min(sapply(samples, min))
  grid.max = max(sapply(samples, max))
  grid = seq(grid.min, grid.max, length=1000)

  # Calculate the ECDFs
  ## ecdfs = lapply(samples, ??)
  evals = lapply(ecdfs, function(f) f(grid))

  # Plot the true CDF
  ## plot(grid, ??, type="l", col="black", xlab="x", ylab = "P(X <= x)")

  # Plot the ECDFs on top
  n.sizes = length(sizes)
  cols = rainbow(n.sizes)
  for (i in 1:n.sizes) {
    lines(grid, evals[[i]], col=cols[i])
  }
  legend("bottomright", legend=sizes, col=cols, lwd=1)
}
```

Drug effect simulation

We're going to continue studying the drug effect model that was discussed in the "Simulation" lecture. Recall, we suppose that there is a new drug that can be optionally given before chemotherapy. We believe those who aren't given the drug experience a reduction in tumor size of percentage:

$$X_{\text{no drug}} \sim 100 \cdot \text{Exp}(\text{mean} = R), \quad R \sim \text{Unif}(0, 1),$$

whereas those who were given the drug experience a reduction in tumor size of percentage:

$$X_{\text{drug}} \sim 100 \cdot \text{Exp}(\text{mean} = 2).$$

- **2a.** Look the code chunk in the lecture that generated data according to the above model. Write a function around this code, called `simulate.data()`, that takes two arguments: `n`, the sample size (number of subjects in each group), with a default value of 60; and `mu.drug`, the mean for the exponential distribution that defines the drug tumor reduction measurements, with a default value of 2. Your function should return a list with two vectors called `no.drug` and `drug`. Each of these two vectors should have length `n`, containing the percentage reduction in tumor size under the appropriate condition (not taking the drug or taking the drug).
- **2b.** Run your function `simulate.data()` without any arguments (hence, relying on the default values of `n` and `mu.drug`), and store the output in `results1`. Print out the first 6 values in both the `results1$no.drug` and `results1$drug` vectors. Now, run `simulate.data()` again, and store its output in `results2`. Again, print out the first 6 values in both the `results2$no.drug` and `results2$drug` vectors. We have effectively simulated two hypothetical datasets. Note that we shouldn't expect the values from `results1` and `results2` to be the same.
- **2c.** Compute the following three numbers: the absolute difference in the mean values of `no.drug` between `results1` and `results2`, the absolute difference in the mean values of `drug` between `results1` and `results2`, and the absolute difference in mean values of `no.drug` and `drug` in 'results1". Of these three numbers, which one is the largest, and does this make sense?
- **2d.** Now, we want to visualize the simulated data. Fortunately, the code to visualize the data is already provided for you in the "Simulation" lecture. Write a function around this code, called `plot.data()`, that takes just one argument `data`, which is a list with components `drug` and `no.drug`. To be clear, this function should create a single plot, with two overlaid histograms, one for`data$no.drug` (in gray) and one for `data$drug` (in red), with the same 20 bins. It should also overlay a density curve for each histogram in the appropriate colors, and produce a legend. One written, call `plot.data()` on each of `results1`, and on `results2`.
- **2e.** In just one line of code total, generate a new simulated data set using `simulate.data()` where `n=1000` and `mu.drug=1.1`, and plot the results using `plot.data()`. In one or two sentences, explain the differences that you see between this plot and the two you produced in the last problem.
- **2f.** In the next problem, we will be generating many hypothetical data sets to see how many subjects we need to observe a difference between taking the drug and not taking the drug. To prepare for this, write a function called `simulate.difference()`, which takes in the same two arguments as `simulate.data()`, namely `n` and `mu.drug`, with the same default parameters as before. Your function should generate a new data set using `simulate.data()` using the appropriate inputs, and then just return the difference in means of `drug` and `no.drug` (no absolute value). Run this function twice with no arguments (hence, using the default parameters) to see that your function is returning different numbers, and run the function once with `n=1000` and `mu.drug=10`. Print out all three return values. This last value should be substantially larger than the first two.

Running simulations, saving money

For the next few questions, we will work with this hypothetical: suppose we work for a drug company that wants to put this new drug out on the market. In order to get FDA approval, your company must demonstrate that the patients who had the drug had **on average** a reduction in tumor size **at least 100 percent greater than** those who didn't receive the drug, or in math:

$$\overline{X}_{\text{drug}} - \overline{X}_{\text{no drug}} \geq 100.$$

Your drug company wants to spend as little money as possible. They want the smallest number n such that, if they were to run a clinical trial with n patients in each of the drug / no drug groups, they would likely succeed in demonstrating that the effect size (as above) is at least 100. Of course, the result of a clinical trial is random; your drug company is willing to take "likely" to mean **successful with probability 0.95**, i.e., successful in 190 of 200 hypothetical clinical trials (though only 1 will be run in reality).

- **3a.** Following the code sketch provided at the end of the "Simulation" lecture, write a function called `rep.sim()`. This function takes four arguments: `nreps` (the number of repetitions, with default value of 200), `n` and `mu.drug` (the values needed for `simulate.difference()`, with the same defaults as before), and `seed` (with default value `NULL`). Your function should run `simulate.differences()` `nreps` number of times, and then return the number of success, i.e., the number of times that the output of `simulate.difference()` exceeds 100. Demonstrate your function works by using it with `mu.drug=1.5`. Hint: to implement `rep.sim()`, you could use a `for()` loop, as shown in the slides, or if you're interested in trying an alternative route, you could use the `replicate()` function. Check the documentation to understand how the latter function.
- **3b.** Now we investigate the effect of the sample size `n`, fixing `mu.drug` to be 2. For each value of `n` in between 5 and 100 (inclusive), run your function `rep.sim()`. You can do this using a `for()` loop or an apply function. Store the number of success in a vector. Just to be clear: for each sample size in between 5 and 100, you should have a corresponding number of successes. Plot the number of successes versus the sample size, and label the axes appropriately. Based on your simulation, what is the smallest sample size for which the number of successes is 190 or more?
- **3c.** Now suppose your drug company told you they only had enough money to enlist 20 subjects in each of the drug / no drug groups, in their clinical trial. They then asked you the following question: how large would `mu.drug` have to be, the mean proportion of tumor reduction in the drug group, in order to have probability 0.95 of a successful drug trial? Run a simulation, much like your simulation in the last problem, to answer this question. Specifically, similar to before, for each value of the input `mu.drug` in between 0 and 5, in increments of 0.25, run your function `rep.sim()`, with `n=20` and `nreps=200`. Plot the number of successes versus the value of `mu.drug`, and label the axes appropriately. What is the smallest value of `mu.drug` for which the number of successes exceeds 190?
- **3d.** We're going to modify the simulation setup from the last question and see how it changes the results we observe. Here is the new setup:
  - We start with `n=5` subjects (as always, this means 5 subjects with the drug, 5 subjects without the drug).
  - We compute the difference in means between using the drug and not using the drug.
  - If this difference is larger than or equal to 100, we declare success and stop.
  - If the difference is smaller than 100, then we collect 5 new subjects with the drug and 5 new subjects without the drug.
  - Once again, we compute the difference in means between the subjects with the drug and the subjects without the drug, and we declare success if this difference is equal to or larger than 100.
  - We keep incrementing by 5 new subjects with the drug and without the drug until we have a total of 60 subjects with the drug and 60 subjects without the drug.
  - If we *still* do not observe a difference in means larger than 100 at this point, then we declare the a failure.

  Change the functions `simulate.data()`, `simulate.difference()` and `rep.sim()`—whatever necessary—to accommodate this new scheme. Then run this simulation with 200 repetitions with `mu.drug=1.5`, and print out how many success there were. How does this number compare with the result you saw earlier in Q3a? Should it be much different?

AB testing

A common task in modern data science is to analyze the results of an AB test. AB tests are essentially controlled experiments: we obtain data from two different conditions, such as the different versions of a website we want to show to users, to try to determine which condition gives better results.

- **4a.** Write a function to simulate collecting data from an AB test where the responses from the A condition follow a normal distribution with mean `a.mean` and standard deviation `a.sd`, whereas responses from the B condition follow a normal distribution with mean `b.mean` and standard deviation `b.sd`.

  Your function's signature should be `ab.collect(n, a.mean, a.sd, b.mean, b.sd, n)` where `n` is the number of samples to collect from each condition and the other arguments are as described above. Your function should return a list with two named components `a.responses` and `b.responses` which contain the responses for each condition respectively. Try your function out for several values of `a.mean`, `a.sd`, `b.mean`, and `b.sd` and check that the sample means and standard deviations approximately match the appropriate theoretical values.

- **4b.** Write a function `test.at.end(n, a.mean, a.sd, b.mean, b.sd)` which uses your function from Q4a to draw samples of size `n` and then runs a t-test to determine whether there is a significant difference. We'll define this as having a p-value at most 0.05. If there is a significant difference, we return either "A" or "B" for whichever condition has the higher mean. If there isn't no significant difference, we return "Inconclusive". Hint: recall `t.test()`, and examine its output on a trial run to figuure out how to extract the p-value. Run your function with `n=2000`, `a.mean=100`, `a.sd=20`, `b.mean=140`, `b.sd=10` and display the result.

- **4c.** Waiting until you collect all of the samples can take a while. So you instead decide to take the following approach:

  - Every day you collect 100 new observations from each condition.
  - At the end of the day you check whether or not the difference is significant.
  - If the difference is significant you declare the higher response to be the winner.
  - If the difference is not significant you continue onto the next day.
  - As before, if you collect all of the samples without finding a significant different you'll declare the result "Inconclusive".

  Note that this kind of sequential sampling is very common in AB testing. Note also the similarity to what we had you do in Q3d.

  Write a function `test.as.you.go(n.per.day, n.days, a.mean, a.sd,  b.mean, b.sd)` to implement this procedure. Your function should return a list with the winner (or "Inconclusive"), as well and the amount of data you needed to collect.

  Run this function on the same example as before with `n.per.day=100` and `n.days=20` (to match final sample sizes). Do you get the same result? Do you save time collecting data?

- **4d.** In practice, most AB tests won't have a clear winner; instead both conditions A and B will be roughly equivalent. In this case we want to avoid *false positives*: saying there's a difference when there isn't really a difference (with respect to the true distributions). Let's run a simulation that checks the false positive rate of the two testing regimes.

  Setting `a.mean = b.mean = 100`, `a.sd = b.sd = 20`, and retaining the number of samples as in the previous examples conduct 1000 AB experiments using each of previous two setups, in `test.at.end()` and `test.as.you.go()`.

  For each, calculate the number of "A" results, "B" results, and "Inconclusive" results. Is this what you would expect to see—recalling that we are declaring significance if the p-value from the t-test is at most 0.05? Does either method of sampling (all-at-once, or as-you-go) perform better than the other, with respect to controlling false positives? **Challenge:** can you explain the behavior you're seeing, with the sequential sampling?