FIT2086 Studio 11
Simulation Based Statistical Methods

Daniel F. Schmidt

October 9, 2017

Contents

1 Introduction

The aim of Studio 11 is to give you an introduction to simulation-based statistical techniques that have only become possible since the advent of digital computers.

During your Studio session, your demonstrator will go through the answers with you, both on the board and on the projector as appropriate. Any questions you do not complete during the session should be completed out of class before the next Studio. Complete solutions will be released on the Friday after your Studio.

2 Simple Confidence Intervals Using the Bootstrap

To get started, begin by installing the `boot` package in R. This will give us access to a number of useful functions for computing bootstrap confidence intervals. Install the package and then load it using `library(boot)`.

1. We will begin by looking at the small blood pressure dataset that we have examined previously. Load the `bpdata.csv` file into the dataframe `bpdata`. This dataset contains blood pressure and related variable measurements on a small population of men aged 45 to 56 years old.

2. We are interested in estimating the population average value of the body-surface (in m^2) `BSA` variable. First, we will use the standard procedures to obtain an estimate of the population average value and a confidence interval using the t-test. To do this, use

   ```
   t.test(x = bpdata$BSA, conf.level=0.95)
   ```

   This returns the sample mean and the associated 95% confidence interval for the sample mean.

3. Let us now use the non-parametric bootstrap to obtain a "distribution free" estimate of the confidence interval for the sample mean. To do this,

   ```
   bs.mean = boot(bpdata$BSA, function(x,I) { return(mean(x[I])) }, 1000)
   ```

   The first argument is the data we wish to bootstrap resample. The third argument is the number of bootstrap samples to draw. The second argument is a function that generates the statistic of the data that we wish to bootstrap. The `boot()` function requires this function to take two arguments: the first is the data, and the second is a list of the bootstrap indices (which samples from the complete data to use). Because our statistic of interest is so simple, we have used the anonymous function technique to define our simple function, which takes the mean of the resampled data.

   After we have run the above code, the results of bootstrapping are stored in the object `bs.mean`. We can examine this object by typing

   ```
   bs.mean
   ```

   We note that the object contains some basic information. "`original`" is the value of the statistic function (in this case, the sample mean) when computed on the complete data. "`bias`" is the average difference between the original value of the statistic, and the set of bootstrapped statistic values (in this case, it is essentially zero) and "`std.error`" is the estimated standard error of our statistic (in this case, the amount we expected our sample mean to change by if we resampled data from our population).

4. Now we have gathered bootstrap samples for the sample mean, we can calculate a confidence interval. The `boot` package offers several different techniques for turning the bootstrap samples into confidence intervals. We can calculate the 95% CI using the "basic" method

```
boot.ci(bs.mean, 0.95, type="basic")
```

which is based directly on the sample percentiles. We can also calculate a "median unbiased" 95% CI which is based on adjusted percentiles to ensure that the interval is centered on the median:

```
boot.ci(bs.mean, 0.95, type="bca")
```

If the data is roughly symmetric, both of these intervals will be close to each other. How do the two intervals computed by the bootstrap compare to the interval produced by the assumption of normality (i.e., the t-test above)?

5. A strength of the bootstrap is that it can be used to obtain intervals on any statistic, without requiring us to derive closed-form formulas, as in the case of the normal distribution. This is the power of the computational based statistical methods. They replace mathematical cleverness with computational power.

   To demonstrate this, let us consider estimating the average population value of BSA using the median. The sample median is

```
median(bpdata$BSA)
```

   We can use the bootstrap to easily compute a 95% confidence interval for this estimate:

```
bs.med = boot(bpdata$BSA, function(x,I) { return(median(x[I])) }, 1000)
```

   Again, our statistic is very simple so we can use the anonymous function syntax when calling `boot()`. Use the results of our bootstrap to compute a 95% confidence interval for the median using `boot.ci()`, using both the basic and BCA method. Compare

   (a) The sample mean and sample median.

   (b) The bootstrap estimated standard errors for both sample mean and sample median.

   (c) The 95% confidence intervals for sample median and sample mean, estimated using the "basic" method.

6. Once we have run the bootstrap we have access to a large number of realisations of our statistic of interest. This collection mimics the sampling distribution of the statistic – that is, the distribution we would obtain if we repeatedly drew new samples from our population and calculated our statistic for these new samples. We can visualise this bootstrap distribution by using a histogram. Do this for the bootstrap distribution of the sample mean:

```
plot(bs.mean)
```

   We can see that the histogram of the bootstrap samples is roughly normally distributed in appearance. Produce the histogram from the bootstrap samples of the median

```
plot(bs.med)
```

   We see that the distribution of the median is very different from the distribution of the mean, and does not appear to be normally distributed. It exhibits a large spike at around 1.975. Can you explain why this might be the case?

3 Bootstrapping Complex Statistics and Permutation Tests

In this question we will look at how we can use the bootstrap to get confidence intervals for much more complex statistics than the simple statistics we examined in the previous question. We will also look at how to use permutation testing to derive more exact p-values for complicated test statistics. In particular, we will look at using these tools in the context of logistic regression.

1. We will once again use the Pima indians diabetes data as a dataset on which to demonstrate this ideas. Begin by loading `pima.train.csv` into the dataframe `pima.train`, and also download the latest version of `my.prediction.stats.R` from Week 11 Moodle, and load it using

   ```
   source("my.prediction.stats.R")
   ```

   This version returns the statistics in addition to (optionally) displaying them on the console, which is useful for bootstrapping.

2. Let us first fit a logistic model to the Pima indians data

   ```
   fit = glm(DIABETES ~ ., pima.train, family=binomial)
   ```

   We can summarise the goodness-of-fit information of this model by computing the "prediction" statistics on the data we trained the model on using

   ```
   my.pred.stats(predict(fit,pima.train,type="response"), pima.train$DIABETES)
   ```

   As we are computing statistics on the data we used to fit the model, they will be optimistic. It is therefore important to be able to quantify the uncertainty we have about these statistics. This is obviously a difficult task to perform analytically, as the formulas for fitting logistic models, and for quantities such as the AUC are very complex. However, the bootstrap offers a nice solution to this. If we call `my.pred.stats()` with the `display=F` option it instead returns the statistics:

   ```
   rv = my.pred.stats(predict(fit,pima.train,type="response"), pima.train$DIABETES, display=F)
   ```

   If you look at `rv` you will see it contains all the statistics that are reported by `my.pred.stats()` if `display=T` (the default).

3. Let us get a 95% confidence interval on the AUC of our fitted model. This can be interpreted as the range of plausible values of AUC we would expect to obtain if we tested our model on new data from the sample population. To do this we first need to create a simple wrapper function for the `boot()` function that takes the bootstrap data indexes, fits a logistic model, computes

the AUC on the bootstrapped data and returns it.

```
boot.auc = function(formula, data, indices)
{
  # Create a bootstrapped version of our data
  d = data[indices,]

  # Fit a logistic regression to the bootstrapped data
  fit = glm(formula, d, family=binomial)

  # Compute the AUC and return it
  target = as.character(fit$terms[[2]])
  rv = my.pred.stats(predict(fit,d,type="response"), d[,target], display=F)
  return(rv$auc)
}
```

We can then use the `boot()` function to get a bootstrap estimate of the confidence interval for the AUC of our logistic regression model:

```
bs = boot(data=pima.train, statistic=boot.auc, R=1000, formula=DIABETES ~ .)
boot.ci(bs,conf=0.95,type="bca")
plot(bs)
```

We see that the bootstrap estimates plausible values for the AUC of our model on future data is approximately $(0.812, 0.873)$. The numbers will differ slightly due to randomness in the bootstrap sampling; to reduce this variability, we can take the number of bootstrap iterations `R` to be larger, but this will result in longer computation times.

4. Let us test how good our confidence interval for the AUC is; to do this, load the `pima.test.csv` dataset into the dataframe `pima.test` and compute the prediction statistics of our model on this test data

   ```
   my.pred.stats(predict(fit,pima.test,type="response"), pima.test$DIABETES)
   ```

   We can see that the AUC our model achieves on this future data is 0.8162, which is within the confidence interval estimated by the bootstrap. This is quite remarkable, given that all we did was re-sampled from our data as if it were the "population"! Obviously, the larger the training sample, the more reliable the bootstrap confidence intervals will be.

5. Next, let us explore the use of permutation testing. This method allows us to obtain p-values for hypothesis tests, even if the problem we are dealing with does not have a nice mathematical solution (such as in the case of the normal distribution). These p-values have the advantage of being (in theory) more accurate than many of the approximations based on the central limit theorem, as they are not based on any approximations. The drawback is that they are computionally expensive. Look at the file `perm.log.reg()`. This code computes permutation test p-values for the null hypothesis that

$$H_0 : \beta_j = 0$$
$$\text{vs}$$
$$H_A : \beta_j \neq 0$$

for a logistic regression model. Go through each line of the code and try to understand what is going on. The p-values reported by `summary(fit)` are based on central limit theorem approximations, so in theory the permutation p-values will be more accurate, particularly if the sample size is small.

Once you have gone through and understood the code, load it into memory using `source()` and let us try using it.

```
rv = perm.log.reg(DIABETES ~ ., pima.train, R=1000)
```

This performs $1,000$ permutations. The p-values are in `rv$p.values`. Have a look at these p-values. Run the same code again, and examine the p-values again. You will notice that they vary quite considerably, particularly in the case of the `BP` variable, which has a smallish p-value. This is because we have only run our test with a small number of permutations.

6. Run the test again, this time with a much larger number of permutations

```
rv = perm.log.reg(DIABETES ~ ., pima.train, R=10000)
```

This might take a while to complete. Once it is done, first examine the p-values produced by the `glm()` function by using

```
summary(fit)
```

Then examine the p-values in `rv$p.values`. How do the two sets of p-values compare? Why do you think so many of the p-values computed by the permutation test are exactly equal to zero?

7. Finally, we can examine the null distribution of the coefficients using the results from the permutation test. For example, to see the null distribution of the coefficient for `INS`

```
hist(rv$perm.coef[,"INS"])
```

The coefficient for `INS` fitted on the complete data is `fit$coefficients[["INS"]]` which is ≈ 0.00035. Compare this value to the plot of the null distribution, which ranges from around -0.002 to 0.002; we see that the observed coefficient is much smaller than the bulk of the values, which is why the p-value is quite large (≈ 0.75). In contrast, let us look at the null distribution for `BP`

```
hist(rv$perm.coef[,"BP"])
```

The coefficient for `BP` fitted on the complete data is ≈ -0.017. From the histogram, we see that the bulk of the null distribution lies between -0.01 and 0.01, which is why the p-value for `BP` is small (≈ 0.005), as a coefficient as large as -0.017 is unlikely to arise just by chance under the null distribution we have found using the permutation approach.

4 Random Number Generation

In this question you will write some simple code to generate random numbers for the exponential distribution. The exponential distribution with scale parameter α is a simple distribution with a probability distribution function

$$p(y \mid \alpha) = \left(\frac{1}{\alpha}\right) \exp\left(-\frac{y}{\alpha}\right).$$

If a RV follows an exponential distribution with scale parameter α, we say that $Y \sim \text{Exp}(\alpha)$. If $Y \sim \text{Exp}(\alpha)$, then $\mathbb{E}[Y] = \alpha$, and a variance of $\mathbb{V}[Y] = \alpha^2$.

1. The simplest way to generate exponential random variables is through the inverse-CDF sampling method. Write a function `my.rexp(n,alpha)` that returns `n` random numbers sampled from an Exp(α) distribution. See slides 22–23 from Lecture 11 for details on how to implement this sampler. *(hint: you can generate* `n` *random uniformly distributed random numbers in a single line by using* `runif(n)`*)*

2. Once you have implemented this function, we can test to see how well it generates random numbers. To do this, we can generate a very large number of random numbers for say $\alpha = 2$

   ```
   x = my.rexp(n = 10000000, alpha = 2)
   ```

   and compare to see how the sample mean and variance compare to the theoretical mean ($\mathbb{E}[Y] = \alpha$) and variance ($\mathbb{V}[Y] = \alpha^2$). To do this for the mean, for example, use

   ```
   mean(x)
   ```

   We see this is very close to the theoretical quantity 2. Do the same for the sample variance, using the `var`() function; you will also note that they are very close. In fact we can even try this for higher moment statistics such as the skewness and kurtotis. The skewness is based on the third moment (Y^3) and measures how asymmetric the distribution is, with a skewness of zero indicating a symmetric distribution. The kurtosis is based on the fourth moment (Y^4) and measures how "heavy" the tails of the distribution are (how fast they vanish to zero), with a value greater than three indicating a heavier tailed distribution than the normal.

   The theoretical skewness and kurtosis of the exponential distribution are 2 and 9, respectively, irrespective of α. To compute the sample skewness and excess kurtosis, install and load the package `moments`, then use the functions `skewness(x)` and `kurtosis(x)`. You will see that our function `my.rexp()` is clearly doing a good job of sampling from the exponential distribution.

3. Finally, we can visually inspect our samples to see how well their distribution matches the theoretical distribution of the exponential distribution. To do this, first let us produce a histogram representation of the probability density function

   ```
   hist(x,probability=T)
   ```

   The `probability=T` option tells R to plot probabilities on the y-axis, instead of frequencies (counts). Then, we can overlay the theoretical distribution using

   ```
   z = seq(0,30,0.001)
   lines(z,(1/alpha)*exp(-z/alpha),col="red")
   ```

   The plot shows the very clear correspondence of the distribution of our random numbers to the theoretical exponential distribution. You an try re-running these experiments with different values of α to confirm our sampler is working.