HW 6 - Improving R

Kelley Breeze

2022-09-26

Contents

Basic t-test

The t-test makes two important assumptions that we need to account for when creating our test data.

1. The data is a random sample AND

2. The data is sampled from a normal distribution

When these assumptions are met, we can use the t-statistic as our test statistic.

We can use the following rejection rules to control the type I errror (in which we reject the null hypothesis when the null hypothesis is in fact true).

- HA : population mean != population mean0 reject if |tobs| > 0.975 quantile of a t-distribution with n-1 degrees of freedom

- HA : population mean < population mean0 reject if tobs < 0.05 quantile of a t-distribution with n-1 degrees of freedom

- HA : population mean > population mean0 reject if tobs > 0.95 quantile of a t-distribution with n-1 degrees of freedom

Writing Functions to do a t-test

Test statistic function

The code below calculates the test statistic (`tobs`) given a numeric vector (`numericVector`) and the mean value to compare against (`mu0`).

```r
testStatistic<- function(numericVector, mu0){
  vectorMean<- mean(numericVector)
  vectorSD<- sd(numericVector)
  nullHypothesis<- mu0

  tobs<- (vectorMean-nullHypothesis)/(vectorSD/(sqrt(length(numericVector))))

  return(tobs)
}
```

Function to reject the null hypothesis or fail to reject it

The code below will take the output from our `testStatistic` function and determine if we should reject or fail to reject the null hypothesis. Here I have entered a default value for our `significanceLevel` of `0.05`. The user also has the ability to enter information for the `outputTestStatistic`, the `sampleSize`, and the tail (`two`, `left` or `right`) Note that if our `hypothesisTest` function returns `TRUE` that means that we will **reject the null hypothesis**, and that if our `hypothesisTest` function returns `FALSE` we will **fail to reject the null hypothesis.**

```r
hypothesisTest<- function(outputTestStatistic, sampleSize, significanceLevel=0.05, tail){
  degreesOfFreedom<- sampleSize-1

  twoTDQuantile<- qt((1-(significanceLevel/2)), df=degreesOfFreedom)

  leftTDQuantile<- qt(significanceLevel, df=degreesOfFreedom)

  rightTDQuantile<- qt((1-significanceLevel), df=degreesOfFreedom)
  #Two-tailed test
  if(tail=="two"){
    rejectNull<- if(abs(outputTestStatistic)>twoTDQuantile){
      TRUE
    } else {
      FALSE
    }
  }
  #One-tail left selected
  else if(tail=="left"){
    rejectNull<- if(outputTestStatistic<leftTDQuantile){
      TRUE
    } else {
      FALSE
    }
  }
  #One tail right selected
  else if(tail=="right"){
    rejectNull<-if(outputTestStatistic>rightTDQuantile){
      TRUE
    } else {
      FALSE
  }
  }
  return(rejectNull)
  }
```

Testing our `hypothesisTest` Function

Next we are going to test out our `hypothesisTest` function using the built-in `iris` data set. We will run a few hypothesis tests to determine if the true mean…

- sepal length differs from 5.5
- sepal width is greater than 3.5
- petal length is less than 4

Before we begin, let's combine our `testStatistic` and `hypothesisTest` functions into one call with options for `numericVector`, `mu0`, `sampleSize`, `significanceLevel` and `tail`. By combining these into one wrapper function we can automatically generate the correct value for sample size from the `numericVector` that we entered, so we do not have to enter a value for `sampleSize` here. I have kept the default significance level here at `0.05`.

```
hypothesisTestWrapper<- function(numericVector, mu0, significanceLevel=0.05, tail){
  outputTestStatistic<- testStatistic(numericVector, mu0)
  sampleSize <- length(numericVector)
  reject<-hypothesisTest(outputTestStatistic, sampleSize, significanceLevel, tail)

  return(reject)
}
```

Iris Test 1

Here we are testing if `Sepal.Length` differs from `5.5`. We will use the default `significanceLevel` of `0.05` for this test. Because we are testing if Ha != H0 this is a two tailed test, and we need to enter `tail="two"`. We will also print the mean value for `Sepal.Length` (`test1Mean`)just to double check that the result of our hypothesis test makes sense. The result of this hypothesis test returned `TRUE`, meaning that we should reject the null hypothesis of 5.5 and say that the mean sepal length is not 5.5. This makes sense when we look at the calculated mean for our data on sepal length, which has an average of 5.84.

```r
test1<-hypothesisTestWrapper(iris$Sepal.Length, 5.5, tail = "two")

test1Mean<-mean(iris$Sepal.Length)

test1Mean
```

```
## [1] 5.843333
```

```r
test1
```

```
## [1] TRUE
```

Iris Test 2

In the second test we want to determine if we should reject the null hypothesis given for `Sepal.Width` (sepal width > 3.5). We will again use the default value for the significance level, but because we want to know if we should reject the null hypothesis for `Sepal.Width` we will enter `tail="left"`. Because the null hypothesis argues that the average width is greater than 3.5, selecting a left tailed test will help us to identify if the true value is **lower** than 3.5. When we run this function the result of our hypothesis test is `TRUE`, which means that we reject our null hypothesis that sepal width is greater than 3.5. This also makes sense when we look at the `test2Mean` of `3.057` which is consistent with rejecting the null hypothesis.

```r
test2<-hypothesisTestWrapper(iris$Sepal.Width, 3.5, tail="left")

test2Mean<- mean(iris$Sepal.Width)

test2Mean
```

```
## [1] 3.057333
```

```r
test2
```

```
## [1] TRUE
```

Iris Test 3

In the final test of our function we will determine if we should reject the null hypothesis that `Petal.Length` is less than `4`. This time we are going to reject the null hypothesis if our data show that `Petal.Length` is actually greater than 4, which means that this is a right tailed test, and we need to enter `tail="right"` when running our function. Here we can see that the returned result is `FALSE`, which means we fail to reject the null hypothesis that `Petal.Length` is less than 4. This makes sense when we check the average petal length of our data, which is 3.758.

```r
test3<-hypothesisTestWrapper(iris$Petal.Length, 4, tail="right")

test3Mean<- mean(iris$Petal.Length)

test3Mean
```

```
## [1] 3.758
```

```
test3
```

```
## [1] FALSE
```

Monte Carlo Study

A Monte Carlo Simulation is one where we generate (pseudo) random values using a random number generator and use those random values to judge properties of tests, intervals, algorithms, etc. We will first generate data from a gamma distribution using different shape parameters and sample sizes. We will then use this generated data to apply the t-test functions that we created above. The hypothesis test is based around the assumption of a normally distributed data, so we will be able to see what impact a different distribution can have on our results.

Pseudocode:

- generate a random sample of size n from the gamma distribution with given `shape` and `rate` parameters specified (see `rgamma()`)
- find the test statistic using the actual mean of the randomly generated data for mu0 (i.e mu0 = shape*rate, implying the null hypothesis H0 is true)
- using the given alternative (`tail` = `left`, `right`, or `two`) determine whether you reject or not
- repeat many times
- observe the proportion of rejected null hypothesis from your repetitions (these are all incorrectly rejected since the null hypothesis is true). This proportion is our Monte Carlo estimate of the alpha value under this data generating process.

Monte Carlo Function

We will first create a function called `monteCarlo` which will generate our gamma distribution data by allowing the user to specify the `sampleSize` desired in each trial, as well as the `shape` and `rate` for the gamma distribution. The function will always replicate `10000` times, for a two-sided test. It will return `alpha`, which is a value calculated based on the percentage of `TRUE` values generated in our hypothesis test. Remember, the `hypothesisTestWrapper` function returns a value of `TRUE` when we should reject our null hypothesis.

```
monteCarlo<- function(sampleSize, shape, rate=1){
  set.seed(123)
    alphaTF<- replicate(10000, expr = {
      gammaSample<-rgamma(sampleSize, shape, rate)
      gammaHypothsis<- hypothesisTestWrapper(gammaSample, mu0=shape*rate, tail = "two")
})
  misclass<- sum(alphaTF==TRUE)
  alpha<-misclass/10000

  return(alpha)
}
```

Now let's enter the values for our specific study

We will use a `shape = 0.5` and a `rate = 1` with a sample size of `n = 10` for a **two-sided test**. We will save the output as an object called `alphaTest1`. As we can see the actual alpha for our test is `0.1376`. This value tells us that in our study we rejected the null hypothesis when it was actually true (Type 1 error) 13.76% of the time.

```
alphaTest1<- monteCarlo(10, 0.5, 1)

alphaTest1
```

```
## [1] 0.1388
```

Parallel Computing

Next we will look at how we can run the above Monte Carlo simulation for several settings of sample size, shape, and rate parameter. The first thing that we want to do is to create vectors for sample size, shape, and rate values as listed below:

- sample size values `size` (10, 20, 30, 50, 100)
- shape values `shape` (0.5, 1, 2, 5, 10, 20)
- rate vector with a value of 1

```
size<- c(10, 20, 30, 50, 100)
shape<- c(0.5, 1, 2, 5, 10, 20)
rate<- 1
```

Cores Setup

The next thing that we need to do is to set up our cores for parallel computing.

```
cores<- detectCores()
cores
```

```
## [1] 8
```

```
cluster<- makeCluster(cores-1)
cluster
```

```
## socket cluster with 7 nodes on host 'localhost'
```

Generating Our List to Parallelize Our Computations Over

After we have our cores set up, we want to set up our `Xlist` list object that we will parallelize our computations over. This list will contain every combination of sample size and shape parameter from the vectors above.

```r
#Create a data frame of all combinations of the size and shape vectors.
expanded<-expand.grid(size, shape)

#Create a function that will take our expanded data frame and return a named list object when used in c
xfun<- function(x){
  return(setNames(as.list(x), nm=c("n", "shape")))
}


Xlist<-apply(expanded, 1, FUN = xfun)

#Let's print the first 6 list elements to make sure everything looks good.
head(Xlist)
```

```
## [[1]]
## [[1]]$n
## [1] 10
##
## [[1]]$shape
## [1] 0.5
##
##
## [[2]]
## [[2]]$n
## [1] 20
##
## [[2]]$shape
## [1] 0.5
##
##
## [[3]]
## [[3]]$n
## [1] 30
##
## [[3]]$shape
## [1] 0.5
##
##
## [[4]]
## [[4]]$n
## [1] 50
##
## [[4]]$shape
## [1] 0.5
##
##
## [[5]]
## [[5]]$n
## [1] 100
##
## [[5]]$shape
## [1] 0.5
##
```

```
##
## [[6]]
## [[6]]$n
## [1] 10
##
## [[6]]$shape
## [1] 1
```

gammaRep Function

Now we will write a function that will take in a list, carry out the replication of gamma samples, find the decisions for each sample, and calculate and return the proportion with type 1 error (alpha) for 10,000 repetitions. This function can then be passed to our `parLapply()` function call to generate our alpha values calculated for each size `n` and `shape` combination for our gamma distributions. The `gammaRep` function will do the following:

1. Generate a gamma distribution based on the size (`n`) and `shape` values specified. This will be a vector of length `n`.

2. Calculate and return the `mean` for the `gammaDis` vector. This should be a single numeric value `gammaAvg`.

3. Find the `tobs` based on the `gammaAvg` found in using the `gammaDisAvg` function. Mu0 is calculated for this data by multiplying the shape and rate value for our gamma distribution. Since the rate value for the entire data set is always equal to one, our Mu0 value will be equal to our shape*1, or just `shape`.

4. We will then use our `tObs` value to perform our hypothesis test. For our hypothesis test if the result returns `TRUE` we should reject the null hypothesis.

5. We will then use the `replicate()` function to repeat steps 1-4 10,000 times. We will store this result within our `gammaRep` function as `alphaTF`.

6. Lastly, we want to calculate the `alpha` value for our set of 10,000 repetitions. We can do this by finding the number of times TRUE was returned in `alphaTF` and divide that by the number of repetitions (10,000).

7. The `gammaRep` function will take in a list object and return the `alpha` value for a given combination of `n` and `shape` values.

```r
gammaRep<- function(x){
  set.seed(123)
      alphaTF<- replicate(10000, expr = {
        gammaDisVec<- rgamma(x$n, x$shape, rate=1)
        gammaAvg<- mean(gammaDisVec)
        tObs<- (gammaAvg-x$shape)/(sd(gammaDisVec)/sqrt(x$n))
        degreesOfFreedom<- (x$n)-1
        twoTDQuantile<- qt((0.975), df=degreesOfFreedom)
        rejectNull<- if(abs(tObs)>twoTDQuantile){
          TRUE
          }
        else {
          FALSE
        }
})
```

```
  misclass<- sum(alphaTF==TRUE)
  alpha<-misclass/10000

  return(setNames(list(x$n, x$shape, alpha), c("n", "shape", "alpha")))
}
```

Using `parLapply()` to generate `alpha` values!

We are now ready to pass the `gammaRep` function to `parLapply` to generate the `alpha` values for each combination of `n` and `shape` values found in our list object called `Xlist`. The results are returned in the form of a list of 30 possible combinations of `n` and `shape` and the corresponding `alpha` value for each. We can see what this looks like by saving the output as an object called `gammaResultsList`. We can print the first 6 list elements using the `head()` function to see what everything looks like in our `gammaResultList`.

```
gammaResultsList<-parLapply(cluster, X=Xlist, gammaRep)

head(gammaResultsList)
```

```
## [[1]]
## [[1]]$n
## [1] 10
##
## [[1]]$shape
## [1] 0.5
##
## [[1]]$alpha
## [1] 0.1388
##
##
## [[2]]
## [[2]]$n
## [1] 20
##
## [[2]]$shape
## [1] 0.5
##
## [[2]]$alpha
## [1] 0.1033
##
##
## [[3]]
## [[3]]$n
## [1] 30
##
## [[3]]$shape
## [1] 0.5
##
## [[3]]$alpha
## [1] 0.0874
##
##
## [[4]]
## [[4]]$n
## [1] 50
##
## [[4]]$shape
## [1] 0.5
##
## [[4]]$alpha
## [1] 0.0783
##
##
## [[5]]
## [[5]]$n
## [1] 100
##
## [[5]]$shape
## [1] 0.5
##
## [[5]]$alpha
## [1] 0.0677
##
##
## [[6]]
## [[6]]$n
## [1] 10
##
## [[6]]$shape
## [1] 1
##
## [[6]]$alpha
## [1] 0.0959
```

Cleaning Up Our Results Output

We now have `alpha` values for all combinations of `n` and `shape`, but the output (`gammaResultsList`) is not in a very reader-friendly form. Let's fix this by converting this list into a data frame and giving some descriptive column names, converting the data frame to a tibble, and then using `pivot_wider()` to generate a table that is easy to read and understand. NOTE: By using `flatten(x)` we are able to preserve our column types. If we did not use `flatten` we would lose this information and all of our columns would return as character vectors. The resulting tibble, `gammaResultsTibble` is printed below.

```
columnNames<- c("n", "shape", "alpha")

gammaResultsTibble<- gammaResultsList%>%
  map_dfr(function(x){
    x = flatten(x); x = set_names(x, columnNames); x
  })%>%
  as_tibble()%>%
  pivot_wider(
    names_from = shape,
    names_prefix = "shape=",
    values_from = alpha)

gammaResultsTibble
```

```
## # A tibble: 5 x 7
##       n `shape=0.5` `shape=1` `shape=2` `shape=5` `shape=10` `shape=20`
##   <dbl>       <dbl>     <dbl>     <dbl>     <dbl>      <dbl>      <dbl>
## 1    10      0.139     0.0959    0.0751    0.0631     0.0528     0.0571
## 2    20      0.103     0.0782    0.0678    0.056      0.0551     0.0541
## 3    30      0.0874    0.0688    0.059     0.0513     0.0505     0.047
## 4    50      0.0783    0.0623    0.0547    0.052      0.0512     0.0436
## 5   100      0.0677    0.058     0.0556    0.0506     0.0471     0.0479
```