Untitled

Zhiyuan Yang

10/4/2022

Writing your own functions to do a t-test

Write a function to calculate the test statistic.

```
library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 4.1.3
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --
## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble  3.1.8     v dplyr   1.0.9
## v tidyr   1.2.0     v stringr 1.4.1
## v readr   2.1.2     v forcats 0.5.2
```

```
## Warning: package 'ggplot2' was built under R version 4.1.3
```

```
## Warning: package 'tibble' was built under R version 4.1.3
```

```
## Warning: package 'tidyr' was built under R version 4.1.3
```

```
## Warning: package 'readr' was built under R version 4.1.3
```

```
## Warning: package 'dplyr' was built under R version 4.1.3
```

```
## Warning: package 'stringr' was built under R version 4.1.3
```

```
## Warning: package 'forcats' was built under R version 4.1.3
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
test_statistic <- function(x1, mu0){

  x_bar <- mean(x1)

  sd_x <- sd(x1)
```

```
  n <- length(x1)

  test_statistic <- (x_bar-mu0)/(sd_x/sqrt(n))

  return(test_statistic)
}
```

comments:

We know the t-statistics definition, which is The one-sample t-test is a statistical hypothesis test used to determine whether an unknown population mean is different from a specific value. Thus, I plan to make two parameter in my function. we need to calculate the sample mean. Thus I used x_bar to present my sample mean. And then I need to calculate the standard deviation, I can use sd() to count it directly. I used sd_x to present the standard deviation. Next, i need to count the number of sample observations, so I used length() to count that. After that, I combine them together to get my t test equation, which will calculated t obs value.

Write a function to determine whether you reject h0 for fail to reject it.

Inputs should be test statistics value, the sample size, the significance level, and the direction of the alternative hypotheis. The output should be a true or false value depending on whether or not you reject the null hypothesis.

```
determine_hy <- function(test_statistic, n, alpha = 0.05, side){

  if(side == "left"){


    #mu < mu0 reject if t_obs <0.05
    quant <- qt( p = alpha, df = n-1)
    return(test_statistic < quant)

  } else if (side == "right"){

    #mu >mu0 reject if t_obs >0.95
    quant <- qt( p = 1-alpha, df = n-1)
    return(test_statistic > quant)

  } else if (side == "two"){


    #mu not equal mu0 reject if abs(t_obs) >0.975
    quant <- qt( p= (1-alpha/2), df = n-1)
    return(abs(test_statistic) > quant)
  }

}
```

Comments:

Since qt() will used to return the inverse probability cumulative density of the Student t-distribution. Thus, I need to consider different conditions, which are one side distribution and two side distribution. I used help(qt) to see how qt() works. The p is vector of probabilities, the df is degrees of freedom. Thus, I prepare make sure what I need to put inside of the qt(). For the probabilities, I know it is related to alpha. Normally, alpha is 0.05. Thus, I will set 0.05 as my default alpha value. If the side is right, we know the probabilities equals 1 - alpha. If the side is left, we know the probabilities equals alpha. If the side is two side, the probabilities is (1-alpha/2). Also, the degree freedom is number of obs-1. I still use n to represent my number of obs. Thus, the df is n-1. Since it has three different condition, I consider use if else statement to show difference condition. In my return statement in the function, I can compare t_obs(called test_stat in my function, which is my first function that used to count t_stat) with quantile of a t-distribution(called quant in my function).

Test how well your function works! Use the built-in iris data set and run a few hypothesis tests. Using the data, determine if the true mean

sepal length differs from 5.5 (i.e. test H0 : mu = 5.5 vs HA : mu not equal 5.5 and report whether or not we reject H0)

```
sepal_length_iris <- iris$Sepal.Length
determine_hy(test_statistic = test_statistic(sepal_length_iris, mu0 = 5.5), side="two", n=length(sepal_l
```

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

Comments:

In this test, it is a two side test since the mu not equal, so my side is two. Based on the null hypothesis, we know mu0 is 5.5.

sepal width is greater than 3.5

```
sepal_Width_iris <- iris$Sepal.Width
determine_hy(test_statistic = test_statistic(sepal_Width_iris, mu0 = 3.5), side="right", n=length(sepal
```

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

Comments:

In this test, It is a one side test since the width is greater than 3.5, so my side is right. Based on the null hypothesis, we know mu0 is 3.5.

petal length is less than 4

```
petal_length_iris <- iris$Petal.Length
determine_hy(test_statistic = test_statistic(petal_length_iris, mu0 = 4), side="left", n=length(petal_le
```

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

Comments: In this test, it is a one side test since the width is less than 3.5, so my side is left. Based on the null hypothesis, we know mu0 is 4.

Quick Monte Carlo study

Write code to do the above when sampling from a gamma distribution with shape = 0.5 and rate = 1 with a sample size of n = 10 for a two-sided test. Use the replicate() function (a wrapper for the sapply() function) to do the data generation and determination of reject/fail to reject with relative ease (with the number of replications being 10000) Then find the mean of the resulting TRUE/FALSE values as your estimated level under these assumptions.

```
set.seed(326)
shape <- 0.5
rate <- 1
n <- 10

wrapper <- replicate(10000, {
  gamma_f <- rgamma(n = 10, shape = 0.5, rate = 1)
  T_stat <- test_statistic(gamma_f, 0.5)
  return(determine_hy(T_stat, 10, 0.05, side = "two"))
})

# find the mean
mean(as.logical(wrapper))
```

```
## [1] 0.1423
```

Commments:

The first thing is that I need to check what kinds of paramaters that I can use in replicate function. The main parameter for this function is n and expr. n is the number of replications, and the expr is the expression to evaluate repeatedly. Based on the question, I know the number of replications is 10000. The next step is that I need to find how to evaluate repeatedly. I decide to use the gamma distribution function. I searched on google, I decide to use the rgamma(). To used this function, I need to know what is shape and scale parameters. They must be positive. Based on the question, I set my shape is 0.5 & set rate as 1, and sample side is 10. Also, I know it is two sided test. Then, I need to calculate the t statistics for this gamma distribution. After searched online, I know I need to find mean for gamma distribution, which will be my mu0. the mu is shape times rate. I applied my gamma function into my t test statistical function. In my return statement, I returned my determine whether you reject H0 function. Finally, I use mean() to find the mean.

Parallel Computing

```
#create a vector of sample size values (10, 20, 30, 50, 100)

sample_size <- c(10, 20, 30, 50, 100)

#create a vector of shape values (0.5, 1, 2, 5, 10, 20)
shape_value <- c(0.5, 1, 2, 5, 10, 20)

#create a rate vector with the value 1
rate_vector <- 1

library(parallel)
```

```r
#our cluster set up via makeCluster()
cores <- detectCores()
cluster <- makeCluster(cores - 1)

#use clusterExport() function to pass a list of our functions that we created
clusterExport(cluster, list("test_statistic", "determine_hy"))

#X a list we will parallelize our computations over (each list element would be a combination of sample
#size and shape parameter)
X <- expand.grid(n = sample_size, shape = shape_value, rate = rate_vector)

combination_list <- list()

for (i in 1:length(X$n)){
  combination_list[[i]] <- c("n" = X$n[i], "shape" = X$shape[i], "rate" = X$rate[i])
}

make_combination <- lapply(combination_list, as.list)

make_combination
```

```
## [[1]]
## [[1]]$n
## [1] 10
##
## [[1]]$shape
## [1] 0.5
##
## [[1]]$rate
## [1] 1
##
##
## [[2]]
## [[2]]$n
## [1] 20
##
## [[2]]$shape
## [1] 0.5
##
## [[2]]$rate
## [1] 1
##
##
## [[3]]
## [[3]]$n
## [1] 30
##
## [[3]]$shape
## [1] 0.5
##
## [[3]]$rate
## [1] 1
##
```

```
##
## [[4]]
## [[4]]$n
## [1] 50
##
## [[4]]$shape
## [1] 0.5
##
## [[4]]$rate
## [1] 1
##
##
## [[5]]
## [[5]]$n
## [1] 100
##
## [[5]]$shape
## [1] 0.5
##
## [[5]]$rate
## [1] 1
##
##
## [[6]]
## [[6]]$n
## [1] 10
##
## [[6]]$shape
## [1] 1
##
## [[6]]$rate
## [1] 1
##
##
## [[7]]
## [[7]]$n
## [1] 20
##
## [[7]]$shape
## [1] 1
##
## [[7]]$rate
## [1] 1
##
##
## [[8]]
## [[8]]$n
## [1] 30
##
## [[8]]$shape
## [1] 1
##
## [[8]]$rate
## [1] 1
```

```
##
##
## [[9]]
## [[9]]$n
## [1] 50
##
## [[9]]$shape
## [1] 1
##
## [[9]]$rate
## [1] 1
##
##
## [[10]]
## [[10]]$n
## [1] 100
##
## [[10]]$shape
## [1] 1
##
## [[10]]$rate
## [1] 1
##
##
## [[11]]
## [[11]]$n
## [1] 10
##
## [[11]]$shape
## [1] 2
##
## [[11]]$rate
## [1] 1
##
##
## [[12]]
## [[12]]$n
## [1] 20
##
## [[12]]$shape
## [1] 2
##
## [[12]]$rate
## [1] 1
##
##
## [[13]]
## [[13]]$n
## [1] 30
##
## [[13]]$shape
## [1] 2
##
## [[13]]$rate
```

```
## [1] 1
##
##
## [[14]]
## [[14]]$n
## [1] 50
##
## [[14]]$shape
## [1] 2
##
## [[14]]$rate
## [1] 1
##
##
## [[15]]
## [[15]]$n
## [1] 100
##
## [[15]]$shape
## [1] 2
##
## [[15]]$rate
## [1] 1
##
##
## [[16]]
## [[16]]$n
## [1] 10
##
## [[16]]$shape
## [1] 5
##
## [[16]]$rate
## [1] 1
##
##
## [[17]]
## [[17]]$n
## [1] 20
##
## [[17]]$shape
## [1] 5
##
## [[17]]$rate
## [1] 1
##
##
## [[18]]
## [[18]]$n
## [1] 30
##
## [[18]]$shape
## [1] 5
##
```

```
## [[18]]$rate
## [1] 1
##
##
## [[19]]
## [[19]]$n
## [1] 50
##
## [[19]]$shape
## [1] 5
##
## [[19]]$rate
## [1] 1
##
##
## [[20]]
## [[20]]$n
## [1] 100
##
## [[20]]$shape
## [1] 5
##
## [[20]]$rate
## [1] 1
##
##
## [[21]]
## [[21]]$n
## [1] 10
##
## [[21]]$shape
## [1] 10
##
## [[21]]$rate
## [1] 1
##
##
## [[22]]
## [[22]]$n
## [1] 20
##
## [[22]]$shape
## [1] 10
##
## [[22]]$rate
## [1] 1
##
##
## [[23]]
## [[23]]$n
## [1] 30
##
## [[23]]$shape
## [1] 10
```

```
##
## [[23]]$rate
## [1] 1
##
##
## [[24]]
## [[24]]$n
## [1] 50
##
## [[24]]$shape
## [1] 10
##
## [[24]]$rate
## [1] 1
##
##
## [[25]]
## [[25]]$n
## [1] 100
##
## [[25]]$shape
## [1] 10
##
## [[25]]$rate
## [1] 1
##
##
## [[26]]
## [[26]]$n
## [1] 10
##
## [[26]]$shape
## [1] 20
##
## [[26]]$rate
## [1] 1
##
##
## [[27]]
## [[27]]$n
## [1] 20
##
## [[27]]$shape
## [1] 20
##
## [[27]]$rate
## [1] 1
##
##
## [[28]]
## [[28]]$n
## [1] 30
##
## [[28]]$shape
```

```
## [1] 20
##
## [[28]]$rate
## [1] 1
##
##
## [[29]]
## [[29]]$n
## [1] 50
##
## [[29]]$shape
## [1] 20
##
## [[29]]$rate
## [1] 1
##
##
## [[30]]
## [[30]]$n
## [1] 100
##
## [[30]]$shape
## [1] 20
##
## [[30]]$rate
## [1] 1
```

```
#The fun function makes the replication of gamma samples, finds the decisions for each sample, and
#calculates and returns the proportion

fun <- function(N, all_comb, alpha, side){
  n <- all_comb$n
  shape <- all_comb$shape
  rate <- all_comb$rate
  All_Mean <- mean(as.logical(replicate(N, {gamma_function <- rgamma(n, shape, rate)
                                            tstat_function <- test_statistic(gamma_function, shape*rate)

     return(determine_hy(tstat_function, n, alpha, side = "two"))
    })))

  return(mean(All_Mean))
}



MyFinalMeans <- parLapply(cluster, N = 10000, make_combination, fun, alpha = 0.05, side = "two")



mean_longer <- tibble(mean = unlist(MyFinalMeans),
                      shape = rep(c("shape = 0.5", "shape = 1", "shape = 2",
                                    "shape = 5", "shape = 10", "shape = 20"),
                      times = 5))
```

```
mean_longer
```

```
## # A tibble: 30 x 2
##      mean shape
##     <dbl> <chr>
##  1 0.141  shape = 0.5
##  2 0.108  shape = 1
##  3 0.093  shape = 2
##  4 0.0811 shape = 5
##  5 0.0652 shape = 10
##  6 0.098  shape = 20
##  7 0.0757 shape = 0.5
##  8 0.0722 shape = 1
##  9 0.0672 shape = 2
## 10 0.0579 shape = 5
## # ... with 20 more rows
```

```
# print out a table giving the results in a reasonably palatable manner
mean_wider <- mean_longer %>% group_by(shape) %>%
              mutate(n = sample_size) %>%
              pivot_wider(names_from = shape, values_from = mean)

mean_wider
```

```
## # A tibble: 5 x 7
##       n `shape = 0.5` `shape = 1` `shape = 2` `shape = 5` `shape = 10` shape =~1
##   <dbl>         <dbl>       <dbl>       <dbl>       <dbl>        <dbl>     <dbl>
## 1    10        0.141       0.108       0.093       0.0811       0.0652    0.098
## 2    20        0.0757      0.0722      0.0672      0.0579       0.0728    0.0675
## 3    30        0.0629      0.0593      0.0551      0.0657       0.0568    0.0578
## 4    50        0.0485      0.0534      0.054       0.0535       0.0546    0.0522
## 5   100        0.052       0.0501      0.0518      0.0524       0.0517    0.0508
## # ... with abbreviated variable name 1: `shape = 20`
```

Comments:

Firstly, I created a function called cores and I set up my cluster by using makeCluster(). I used clusterExport() function to pass a list of my core function. I used parLapply() to get my final means. The next step, I need to get my X, X a list that I will parallelize my computations over. Since each list element would be a combination of sample size and shape parameter. I used expand.grid to created my X, and then used a for loop to create the combination list. And then, I created my fun function, which just include my replicate() and mean() code. After I did those steps, I can use parLapply() to execute the above Monte Carlo study for combinations of sample sizes and shape values. After I get my final means, I need to print out a table giving the results in a reasonably palatable manner. I need to give the col name, so in my mean_longer, I gave them name. However, I need to use group_by to make it to be a wider version. Thus, I group_by the shape col. Finally, I got the same result as professor result.