Improving R HW

Sneha Karanjai

2022-10-01

The purpose of this homework is to gain practice with writing functions, vectorized functions, and parallel programming.

Setting seed

Since this document involves a lot of randomization and running a function multiple times, it is better to set a seed so that we can replicate the results with each run.

```
set.seed(1234)
```

Basic t-test

t-test is a test to control the probability of controlling α value (Type-I error). The t-test, however, works under an under of number of assumptions that are :

- The data is a random sample
- The data is sampled from a normal distribution

If these assumptions are satisfied, we use t-statistics as our test statistics :

$$t_{obs} = \frac{\bar{y} - \mu_0}{s\sqrt{n}}$$

The below is a simulation study to check how well significance level is controlled at α when it comes from a distribution that is not normal.

Steps to code t-statistics from scratch

We start by writing a custom function to calculate the t-statistics.

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.6     v dplyr   1.0.7
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   2.0.1     v forcats 0.5.1
```

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

```
t_calc <- function(x, mu_0){
  obs_mean = mean(x)
  standard_dev = sd(x)
  sample_size = length(x)
  t_statistics = (obs_mean - mu_0) / (standard_dev / sqrt(sample_size))
  return(t_statistics)
}
```

We now write a function to check if we reject H_0 or fail to reject H_0 based on the rejection rules of a t-test.

```
H0_test <- function(t_statistics, n, direction){
  decision <- ifelse(direction == "two-sided", abs(t_statistics) > qt(0.975, df = n-1),
                     ifelse(direction == "left", t_statistics < qt(0.05, df = n-1),
                            ifelse(direction == "right", t_statistics > qt(0.95, df = n-1), FALSE)))
}
```

Testing

Testing the functions of iris dataset

1. $H_0 : Sepal.Length \neq 5.5$

```
t_val <- t_calc(iris$Sepal.Length, 5.5)
t_val
```

```
## [1] 5.078045
```

```
decision <- H0_test(t_val, n = length(iris$Sepal.Length), direction = "two-sided")
decision
```

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

We reject the null hypothesis that sepal length differs from 5.5.

2. $H_0 : Sepal.Width > 3.5$

```
t_val <- t_calc(iris$Sepal.Width, 3.5)
t_val
```

```
## [1] -12.43853
```

```
decision <- H0_test(t_val, n = length(iris$Sepal.Length), direction = "right")
decision
```

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

We fail to reject the null hypothesis that the Sepal Width greater than 3.5.

3. $H_0 : Petal.Length < 4$

```
t_val <- t_calc(iris$Petal.Length, 4)
t_val
```

```
## [1] -1.67897
```

```
decision <- H0_test(t_val, length(iris$Sepal.Length), direction = "left")
decision
```

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

We reject the null hypothesis that the SPetal Length lesser than 4.

Quick 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'll generate data from a gamma distribution using different shape parameters and sample sizes. We'll then apply the t-test functions from above and see how well the α level is controlled under the incorrect assumption about the distribution our data is generated from.

The type 1 error is usually considered the worse type of error so tests are set up to control the probability of making this type of error (called the significane level). We define

$$\alpha = P(Type\ I\ error) = P(Rejecting\ H_0\ given\ H0\ is\ True)$$

`monte_carlo()` will take the input parameters shape, size, rate, and direction for generating the data from a gamma distribution using the size, shape, and rate. We calculate the population mean as shape*rate. The t-statistics and the decision to reject or fail to reject H_0 using the functions built above.

```
monte_carlo <- function(shape, size, rate, direction){
  x <- rgamma(n=size, shape=shape, rate=rate)
  pop_mean <- shape*rate
  t_val <- t_calc(x, pop_mean)
  decision <- H0_test(t_val, length(x), direction = direction)
}
```

```
results <- replicate(n = 10000, monte_carlo(shape = 0.5, size = 10, rate = 1, "two-sided"))
output <- table(results)
output
```

```
## results
## FALSE  TRUE
##  8641  1359
```

```
mean(output["TRUE"]/output["FALSE"])
```

```
## [1] 0.1572735
```

This proportion is our Monte Carlo estimate of the α value under this data generating process.

Parallel Computing

Now we run the monte carlo simulation for different combination of parameters. We save two vectors for the different sample size and shape values. Following which we do the necessary steps for setting up parallel processing.

```
library(parallel)
n <- c(10, 20, 30, 50, 100)
shape_val <- c(0.5, 1, 2, 5, 10, 20)
rate <- c(1)
cores <- detectCores()
cores
```

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

```
cluster <- makeCluster(cores - 1)
clusterExport(cluster, list("t_calc", "H0_test", "monte_carlo"))
clusterEvalQ(cluster, library(tidyverse))
```

```
## [[1]]
##  [1] "forcats"   "stringr"   "dplyr"     "purrr"     "readr"     "tidyr"
##  [7] "tibble"    "ggplot2"   "tidyverse" "stats"     "graphics"  "grDevices"
## [13] "utils"     "datasets"  "methods"   "base"
##
## [[2]]
##  [1] "forcats"   "stringr"   "dplyr"     "purrr"     "readr"     "tidyr"
##  [7] "tibble"    "ggplot2"   "tidyverse" "stats"     "graphics"  "grDevices"
## [13] "utils"     "datasets"  "methods"   "base"
##
## [[3]]
##  [1] "forcats"   "stringr"   "dplyr"     "purrr"     "readr"     "tidyr"
##  [7] "tibble"    "ggplot2"   "tidyverse" "stats"     "graphics"  "grDevices"
## [13] "utils"     "datasets"  "methods"   "base"
##
## [[4]]
##  [1] "forcats"   "stringr"   "dplyr"     "purrr"     "readr"     "tidyr"
##  [7] "tibble"    "ggplot2"   "tidyverse" "stats"     "graphics"  "grDevices"
## [13] "utils"     "datasets"  "methods"   "base"
##
## [[5]]
```

```
##  [1] "forcats"   "stringr"   "dplyr"     "purrr"     "readr"     "tidyr"
##  [7] "tibble"    "ggplot2"   "tidyverse" "stats"     "graphics"  "grDevices"
## [13] "utils"     "datasets"  "methods"   "base"
##
## [[6]]
##  [1] "forcats"   "stringr"   "dplyr"     "purrr"     "readr"     "tidyr"
##  [7] "tibble"    "ggplot2"   "tidyverse" "stats"     "graphics"  "grDevices"
## [13] "utils"     "datasets"  "methods"   "base"
##
## [[7]]
##  [1] "forcats"   "stringr"   "dplyr"     "purrr"     "readr"     "tidyr"
##  [7] "tibble"    "ggplot2"   "tidyverse" "stats"     "graphics"  "grDevices"
## [13] "utils"     "datasets"  "methods"   "base"
```

We now use `parLapply()` to build a function that takes every combination of the shape and sample size and applies monte carlo simulation to it 10000 times to get a proportion for each of the combination.

```
parallel_results <- parLapply(cluster,
                              X = apply(expand_grid(n, shape_val),
                                        MARGIN = 1, as.list),
                              fun = function(x){
                                results = replicate(n = 10000,
                                                    monte_carlo(x$shape_val,
                                                                x$n,
                                                                rate = 1,
                                                                "two-sided"))
                                output = table(results)
                                proportion = mean(output["TRUE"]/output["FALSE"])
                                proportion
                                })
```

We save the proportion results in a dataframe.

```
df = expand_grid(n, shape_val) %>%
                                  as_tibble() %>%
                                  add_column(values = unlist(parallel_results)) %>%
                                  pivot_wider(names_from = shape_val, values_from = values, names_prefi
df
```

```
## # A tibble: 5 x 7
##       n `shape = 0.5` `shape = 1` `shape = 2` `shape = 5` `shape = 10`
##   <dbl>         <dbl>       <dbl>       <dbl>       <dbl>        <dbl>
## 1    10         0.158       0.107       0.0865      0.0679       0.0593
## 2    20         0.119       0.0874      0.0715      0.0599       0.0608
## 3    30         0.102       0.0768      0.0651      0.0580       0.0545
## 4    50         0.0811      0.0697      0.0591      0.0539       0.0501
## 5   100         0.0775      0.0635      0.0579      0.0517       0.0531
## # ... with 1 more variable: shape = 20 <dbl>
```