Permutation Test

Dustin Smith

2024-02-08

Introduction

I attempted to preform the Permutation test in two different ways. One was to recreate what was used in the notes, and the other was to preform a sort of 'simulation' of the shuffling to generate a p-value. Both turned out to be quite close.

True Permutaiton Test.

I created two general functions to find compute a pulled t-test, and find measures (being the mean, sample size, and standard deviation) of the responses' iterations. They are below. As I did not want to create a general function, I hard coded the degree of freedom.

```
t_test <- function(means,sigma,n){
  df <- 5
  standard_pooled <- ((n[1] - 1) * (sigma[1] ^ 2) + (n[2] - 1) * (sigma[2] ^ 2 ))/(n[1]
+ n[2] - 2)
  standard_error_combined <- sqrt(standard_pooled) * sqrt( 1 / n[1] + 1 / n[2])

  # The produces the initial statistics
  t_statistic <- (diff(-means) - 0) / standard_error_combined
  treatement_diffs <- diff(-means)

  return(data.frame(t_statistic = t_statistic, treatement_diffs = treatement_diffs))
}

## This function will produce the means and sds of each treatment type.
find_measures <- function(data, treatments){
  for(i in 1:treatments){
    # Here I am assigning measures to each treatment.
    means[i] <- mean(data$response[which(data$treatment == i)])
    sigma[i] <- sd(data$response[which(data$treatment == i)])
    n[i] <- length(data$response[which(data$treatment == i)])
  }

  return(data.frame(means = means, sigma = sigma, n = n))
}
```

From here I created several empty place holder objects, and computed the original experiment's measures and t-test. You can see the initial difference and test value below.

```
data <- data.frame(treatment = c(2,2,1,1,2,1,1), response = c(14,16,19,17,15,13,17))

means <- c(0,0)
sigma <- c(0,0)
n <- c(0,0)

## Calculate initial t-test of given sample
```

```
  measures <- find_measures(data,treatments = 2)
  perm_table <- t_test(means = measures$means, sigma = measures$sigma, n = measures$n)
  (perm_table)

##   t_statistic treatement_diffs
## 1   0.9583148              1.5
```

The next task was to create the permutations of the shuffled treatment values (1,1,1,1,2,2,2). At first I attempted to recreate Heap's Algorithm. I used remainders to switch values in the initial treatment set up. That is, if, by dividing the iteration by 7, the remainder was odd I switched the values of the first treatment with the last treatments given they were unequal. If the remainder was even then the first treatment value was swapped with the remainder's value. Throughout the iterations, I kept track of the count to allow for new rows to be created. While this did create many different permutations, it did not seem to create unique ones. My attempt usually ended with a significantly reduced t-statistic.

As opposed to building a check to eliminate repeated permutations, I ended up making the use of expand.grid(). It took me some time of playing with the function until I learn how it worked. In the end, I used lapply() to create a list of seven c(1,2). Using expand.grid() on this list created a large list of unique permutations of 1s and 2s. To filter out the permutations, I found each permutations that added to 10. (That is :$ 4*1* + 2*3* = 10$). This found the desired list of permutations quickly. You can see the iterations that created the correct treatments below. I also changed the data structure to a Matrix for use of my functions above.

```
  perms <- expand.grid(lapply((1:7), function(i) c(2,1)))
  head(perms)

##   Var1 Var2 Var3 Var4 Var5 Var6 Var7
## 1    2    2    2    2    2    2    2
## 2    1    2    2    2    2    2    2
## 3    2    1    2    2    2    2    2
## 4    1    1    2    2    2    2    2
## 5    2    2    1    2    2    2    2
## 6    1    2    1    2    2    2    2

  perms <- perms[which(rowSums(perms) == 10),]
  (perm_data <- (as.matrix(perms)))

##    Var1 Var2 Var3 Var4 Var5 Var6 Var7
## 16    1    1    1    1    2    2    2
## 24    1    1    1    2    1    2    2
## 28    1    1    2    1    1    2    2
## 30    1    2    1    1    1    2    2
## 31    2    1    1    1    1    2    2
## 40    1    1    1    2    2    1    2
## 44    1    1    2    1    2    1    2
## 46    1    2    1    1    2    1    2
## 47    2    1    1    1    2    1    2
## 52    1    1    2    2    1    1    2
## 54    1    2    1    2    1    1    2
## 55    2    1    1    2    1    1    2
## 58    1    2    2    1    1    1    2
## 59    2    1    2    1    1    1    2
## 61    2    2    1    1    1    1    2
## 72    1    1    1    2    2    2    1
## 76    1    1    2    1    2    2    1
## 78    1    2    1    1    2    2    1
## 79     2    1    1    1    2    2    1
## 84     1    1    2    2    1    2    1
## 86     1    2    1    2    1    2    1
## 87     2    1    1    2    1    2    1
## 90     1    2    2    1    1    2    1
## 91     2    1    2    1    1    2    1
## 93     2    2    1    1    1    2    1
## 100    1    1    2    2    2    1    1
## 102    1    2    1    2    2    1    1
## 103    2    1    1    2    2    1    1
## 106    1    2    2    1    2    1    1
## 107    2    1    2    1    2    1    1
## 109    2    2    1    1    2    1    1
## 114    1    2    2    2    1    1    1
## 115    2    1    2    2    1    1    1
## 117    2    2    1    2    1    1    1
## 121    2    2    2    1    1    1    1
```

It was then quite easy to find the permutation's test statistic. I created a for loop to run my functions through each treatment permutation and store it. Then I calculated the sum of the t-statistics that were greater then or equal to the initial t-statistic. I found the results below. (I included the statistic provided by the notes for reference).

```
  for(i in 1:nrow(perm_data)){
    data <- data.frame(treatment = perm_data[i,], response = c(14,16,19,17,15,13,17))
    measures <- find_measures(data,treatments = 2)
    perm_table[i+1,] <- t_test(means = measures$means, sigma = measures$sigma, n = measur
es$n)
  }

  count <- 0
  for(i in 2:nrow(perm_table)){ #Not this begins at 2, as iteration one is the original t
o compare.
    if(abs(perm_table$t_statistic[i]) >= perm_table$t_statistic[1]){
      count <- count + 1
    }
  }

  (permutation_statistic <- (count)/(nrow(perm_table) - 1))

## [1] 0.3714286

  2*(1-pt(perm_table[1,1], 5))

## [1] 0.3819156
```

Attempt two

This was actually the first attempt at recreating the Permutation test. But, I thought it would be better to discuss it's use second. For this test, I chose to change how the test was preformed. Instead of creating an exact number of unique permutation, I ran the shuffles through a large amount of iterations. Essentially creating a simple simulation of a true permutation test. This seemed to create a very similar t-statistic. It was, however, slightly smaller.

I ran the initial statistic as I did for the other test.

```
  data <- data.frame(treatment = c(2,2,1,1,2,1,1), response = c(14,16,19,17,15,13,17))

  means <- c(0,0)
  sigma <- c(0,0)
  n <- c(0,0)

  ## Calculate initial t-test of given sample
  measures <- find_measures(data,treatments = 2)
  perm_table <- t_test(means = measures$means, sigma = measures$sigma, n = measures$n)
  (perm_table)

##   t_statistic treatement_diffs
## 1   0.9583148              1.5
```

I then created a 'fixed' data set for the treatments. (I changed the order to 1,1,1,1,2,2,2 for convenience).

```
  fixed_data <- data.frame(treatments = c(rep(1,measures$n[1]),rep(2,measures$n[2])), res
ponse = data$response)
  m <- measures$n[1]
  (fixed_data)

##   treatments response
## 1          1       14
## 2          1       16
## 3          1       19
## 4          1       17
## 5          2       15
## 6          2       13
## 7          2       17
```

I then created a for loop to run a specified number of repetitions. The for loop replaced the "treatments" column with twos, then found a sample of 4 numbers from 1 to 7 and inserted a "1" in that index for the "treatments" column. Then I ran the iteration through my functions and stored the results. I also considered instead fixing the treatments and using sample() on the seven response variables while leaving REPLACE = FALSE to reorder the response values. I believe it would have had the same result. You can see the code and results below. The code was quite slow do to recreating the table each iteration.

```
  reps <- 1000
  for(i in 1:reps){
    fixed_data[,1] <- rep(2,nrow(data))
    # I only sampled the location of the first treatment.
    fixed_data[c(sample(1:nrow(data),m)),1] <- 1
    measures <- find_measures(fixed_data,treatments = 2)
    perm_table[i + 1,] <- t_test(means = measures$means, sigma = measures$sigma, n = meas
ures$n)

  }

  ## This loop is to iterate through the found statistics and find the percentage more ex
treme then the original.
  count <- 0
  for(i in 2:nrow(perm_table)){ #Not this begins at 2, as iteration one is the original t
o compare.
    if(abs(perm_table$t_statistic[i]) >= perm_table$t_statistic[1]){
      count <- count + 1
    }
  }

  (permutation_statistic <- (count)/(nrow(perm_table) - 1))

## [1] 0.399

  2*(1-pt(perm_table[1,1], 5))

## [1] 0.3819156
```

As you can see, the result is quite similar to the original t-test. With several attempts, I found that it is typically less than the original Permutation test, but it was much easier to create and used.

Below is the entire script that I used to run the code.

```
#--- Created By Dustin Smith on 1/25/2024 ---#
#--- Edited on 2/7/2024                   ---#

### This script is used to preform a Permutation Test on a set of samples

### Here are some functions that will be used below:
  ## This is for calculating a t_test
  t_test <- function(means,sigma,n){
    df <- 5
    standard_pooled <- ((n[1] - 1) * (sigma[1] ^ 2) + (n[2] - 1) * (sigma[2] ^ 2 )) / (n[
1] + n[2] - 2)
    standard_error_combined <- sqrt(standard_pooled) * sqrt( 1 / n[1] + 1 / n[2])

    # The produces the initial statistics
    t_statistic <- (diff(-means) - 0) / standard_error_combined
    treatement_diffs <- diff(-means)

    return(data.frame(t_statistic = t_statistic, treatement_diffs = treatement_diffs))
  }

  ## This function will produce the means and sds of each treatment type.
  find_measures <- function(data, treatments){
    for(i in 1:treatments){
      # Here I am assigning measures to each treatment.
      means[i] <- mean(data$response[which(data$treatment == i)])
      sigma[i] <- sd(data$response[which(data$treatment == i)])
      n[i] <- length(data$response[which(data$treatment == i)])
    }

    return(data.frame(means = means, sigma = sigma, n = n))
  }
```

```r
####------------------------ Attempt 1 --------------------------------####

  ## I am setting initial values here
  data <- data.frame(treatment = c(2,2,1,1,2,1,1), response = c(14,16,19,17,15,13,17))

  means <- c(0,0)
  sigma <- c(0,0)
  n <- c(0,0)

  ## Calculate initial t-test of given sample
  measures <- find_measures(data,treatments = 2)
  perm_table <- t_test(means = measures$means, sigma = measures$sigma, n = measures$n)

  ## Preform "Boot-legging" of given sample, by reshuffling the values and producing new
t-values
  fixed_data <- data.frame(treatments = c(rep(1,measures$n[1]),rep(2,measures$n[2])), res
ponse = data$response)
  m <- measures$n[1]

  ## This function is designed to shuffle the placements, and find the new test measures
  reps <- 1000
  for(i in 1:reps){
    fixed_data[,1] <- rep(2,nrow(data))
    # I only sampled the location of the first treatment.
    fixed_data[c(sample(1:nrow(data),m)),1] <- 1
    measures <- find_measures(fixed_data,treatments = 2)
    perm_table[i + 1,] <- t_test(means = measures$means, sigma = measures$sigma, n = meas
ures$n)

  }

  ## This loop is to iterate through the found statistics and find the percentage more ex
treme then the original.
  count <- 0
  for(i in 2:nrow(perm_table)){ #Not this begins at 2, as iteration one is the original t
o compare.
    if(abs(perm_table$t_statistic[i]) >= perm_table$t_statistic[1]){
      count <- count + 1
    }
  }

  (permutation_statistic <- (count)/(nrow(perm_table) - 1))

  2*(1-pt(perm_table[1,1], 5))




  ####------------------------ Attempt 2 --------------------------------####

  ## I am setting initial values here
  data <- data.frame(treatment = c(2,2,1,1,2,1,1), response = c(14,16,19,17,15,13,17))

  means <- c(0,0)
  sigma <- c(0,0)
  n <- c(0,0)

  ## Calculate initial t-test of given sample
  measures <- find_measures(data,treatments = 2)
  perm_table <- t_test(means = measures$means, sigma = measures$sigma, n = measures$n)

  ## The below method should find all possible permutations of a list of 7 ones and 7 two
s.
  ## It is however not restricted to exactly 4 ones and 3 twos.
  perms <- expand.grid(lapply((1:7), function(i) c(2,1)))
  ## Here I have filtered out the permutations to the ones that contain the desired 4 one
s, and
  ## 3 twos. It does so by testing the sums to be 4 + 3*2 = 10.
  perms <- perms[which(rowSums(perms) == 10),]
  perm_data <- (as.matrix(perms))

  ## Compare test value to two-way t-test of original experiment.
  for(i in 1:nrow(perm_data)){
    data <- data.frame(treatment = perm_data[i,], response = c(14,16,19,17,15,13,17))
    measures <- find_measures(data,treatments = 2)
    perm_table[i+1,] <- t_test(means = measures$means, sigma = measures$sigma, n = measur
es$n)
  }

  count <- 0
  for(i in 2:nrow(perm_table)){ #Not this begins at 2, as iteration one is the original t
o compare.
    if(abs(perm_table$t_statistic[i]) >= perm_table$t_statistic[1]){
      count <- count + 1
    }
  }

  (permutation_statistic <- (count)/(nrow(perm_table) - 1))

  2*(1-pt(perm_table[1,1], 5))
```