STAT 413/613 HW 3: Lists and Iteration

Yunting Chiu

2020-12-10

Instructions

1. Clone this homework repo to your homework directory as a new repo.
2. Rename the starter file under the analysis directory as `hw_01_yourname.Rmd` and use it for your solutions.
3. Modify the "author" field in the YAML header.
4. Stage and Commit R Markdown and HTML files (no PDF files).
5. **Push both .Rmd and HTML files to GitHub**.

- Make sure you have knitted to HTML prior to staging, committing, and pushing your final submission.

6. **Commit each time you answer a part of question, e.g. 1.1**
7. **Push to GitHub after each major question**
8. When complete, submit a response in Canvas

- Only include necessary code to answer the questions.
- Most of the functions you use should be from the tidyverse. Too much base R will result in point deductions.
- Use Pull requests and or email to ask me any questions. If you email, please ensure your most recent code is pushed to GitHub.
- Learning Outcomes:
  - Manipulate vectors in base-R syntax.
  - Apply iterations with for loops.

Libraries

```
library(tidyverse)
```

```
## -- Attaching packages --------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.3     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0
```

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

```
## x dplyr::lag()    masks stats::lag()
```

```
library(stats)
```

Working with Lists

Because of their generality, lists (or list-like objects) are often the output of many statistical procedures in R. The file `fpout.RDS` in the data folder contains sample output from using fitPoly, a statistical method to quantify properties of locations on the genome.

1. Use `readRDS()` and a relative path to read this data into R.

```
fpout_01 <- readRDS(file = "../data/fpout.RDS")
```

2. Use appropriate functions to obtain the length of the list and then the vector of the names from the list.

```
length(fpout_01)
```

```
## [1] 7
```

```
names(fpout_01)
```

```
## [1] "log"          "modeldata"    "allmodeldata" "scores"       "diploscores"
## [6] "time"         "prop_miss"
```

3. The `diploscores` element does not provide any information. Remove it from the list.
   Revised: - 0.1 Did not remove from the original list but created a new list. Could assign Null to the list element.

```
fpout_01 <- fpout_01[-c(5)]

# check again
length(fpout_01)
```

```
## [1] 6
```

```
names(fpout_01)
```

```
## [1] "log"          "modeldata"    "allmodeldata" "scores"       "time"
## [6] "prop_miss"
```

```
# second method
#fpout$diploscores <- NULL

# third method
#[[5]] <- NULL
```

4. The `scores` element contains the output most users would want. The variables in `scores` called `P0`, `P1`, `P2`, `P3`, `P4`, `P5`, and `P6` contain "posterior probabilities" for each individual for values `0`, `1`, `2`, `3`, `4`, `5`, and `6` (respectively).

- A quantity useful in Bayesian analysis is called the "posterior mean," which in this case is calculated as follows:
  - posterior_mean = `(P0 * 0) + (P1 * 1) + (P2 * 2) + (P3 * 3) + (P4 * 4) + (P5 * 5) + (P6 * 6)`.
- Calculate the posterior mean for each individual and add this as a new variable in the `scores` data frame.
  Revised: 3.1.4: - 0.75 Not added back to the original data frame but to a new data frame outside the list

```
fpout_01$scores %>%
  mutate(Posterior_Mean = (P0 * 0) + (P1 * 1) + (P2 * 2) + (P3 * 3) + (P4 * 4) + (P5 * 5) + (P6 * 6)) ->

head(fpout_01$scores)
```

```
##   marker MarkerName SampleName     ratio P0            P1           P2
## 1      1        SNP          2 0.9450980  0  0.000000e+00 0.000000e+00
## 2      1        SNP          3 0.9186047  0  1.513089e-70 1.630727e-32
## 3      1        SNP          4 0.9976387  0 4.965532e-143 6.960499e-85
## 4      1        SNP          5 1.0000000  0 3.237313e-161 5.236498e-99
## 5      1        SNP          6 0.9202756  0  2.644688e-71 5.145367e-33
## 6      1        SNP          7 0.9037620  0  4.181303e-64 2.965385e-28
##              P3           P4           P5            P6 maxgeno      maxP geno
## 1 5.487698e-278 1.792528e-95 1.000000e+00 1.765903e-145       5 1.0000000    5
## 2  1.688654e-15 8.489692e-05 9.999151e-01  2.727826e-18       5 0.9999151    5
## 3  2.023616e-54 1.173498e-31 8.163299e-14  1.000000e+00       6 1.0000000    6
## 4  7.915965e-66 1.252725e-40 5.265544e-20  1.000000e+00       6 1.0000000    6
## 5  7.908401e-16 5.665944e-05 9.999433e-01  6.710451e-18       5 0.9999433    5
## 6  1.067884e-12 2.639557e-03 9.973604e-01  1.287973e-21       5 0.9973604    5
##   Posterior_Mean
## 1       5.000000
## 2       4.999915
## 3       6.000000
## 4       6.000000
## 5       4.999943
## 6       4.997360
```

5. Use a `map*()` function to identify the names of the variables in the `scores` data frame that are *not* of type double.
   Not correct: 3.1.5: - 0.25 Map function identified all names, not just the names of the variables that were Not double. A very complex approach which could have been done with select, colMeans and assignment back to the lists as a new vector.

```
# keep: keep the TRUE list
# discard: keep the FALSE list
discard(fpout_01$scores, is_double) -> notDouble

head(notDouble)
```

```
##   marker MarkerName SampleName
## 1      1        SNP          2
## 2      1        SNP          3
## 3      1        SNP          4
## 4      1        SNP          5
## 5      1        SNP          6
## 6      1        SNP          7
```

```
# $marker, $MarkerName, and $SampleName, are not dbl type
```

Revised: Is this correct? No – it does not return just the names. Here are some alternates.

```
names(fpout_01$scores)[map_chr(fpout_01$scores, typeof)!= "double"]
```

```
## [1] "marker"     "MarkerName" "SampleName"
```

```
names(keep(fpout_01$scores, ~ typeof(.) != "double"))
```

```
## [1] "marker"     "MarkerName" "SampleName"
```

```
names(discard(fpout_01$scores, ~ typeof(.) == "double"))
```

```
## [1] "marker"     "MarkerName" "SampleName"
```

and a simpler approach to Column Means

```
fpout_01$scores %>%
  select(-marker, -MarkerName, -SampleName) %>%
  colMeans(na.rm = TRUE) ->
  fpout_01$col_means
```

- Create a new element called `col_means` in the list that contains just the column means of all the *double* variables in the `scores` data frame.

```
fpout_01$scores %>%
 select(ratio:Posterior_Mean) -> scores_02
col_means <- list(col_means = map_dbl(scores_02, mean, na.rm = TRUE))

#  Create a new list called `col_means`
print(col_means)
```

```
## $col_means
##          ratio             P0             P1             P2             P3
##   9.110837e-01  1.141826e-236   5.571926e-24   3.159587e-06   1.528993e-02
##             P4             P5             P6        maxgeno           maxP
##   2.535163e-01   5.074307e-01   2.237600e-01   4.951049e+00   9.780702e-01
##          geno Posterior_Mean
##   5.095652e+00   4.939654e+00
```

```
# adding to the existing list to "fpout_02"
append(fpout_01, col_means) -> fpout_02
# test whether is successful or not
length(fpout_02)
```

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

6. Demonstrate three different ways to extract the `col_means` element from the list. The extracted element should *not* be a list.

```
# first way
fpout_02[[7]]
```

```
##          ratio             P0             P1             P2             P3
##   9.110837e-01  1.141826e-236   5.571926e-24   3.159587e-06   1.528993e-02
##             P4             P5             P6        maxgeno           maxP
##   2.535163e-01   5.074307e-01   2.237600e-01   4.951049e+00   9.780702e-01
##          geno Posterior_Mean
##   5.095652e+00   4.939654e+00
```

```
# second way
fpout_02$col_means
```

```
##          ratio             P0             P1             P2             P3
##   9.110837e-01  1.141826e-236   5.571926e-24   3.159587e-06   1.528993e-02
##             P4             P5             P6        maxgeno           maxP
##   2.535163e-01   5.074307e-01   2.237600e-01   4.951049e+00   9.780702e-01
##          geno Posterior_Mean
##   5.095652e+00   4.939654e+00
```

```r
# third method
fpout_02[["col_means"]]
```

```
##          ratio             P0             P1             P2             P3
##   9.110837e-01  1.141826e-236   5.571926e-24   3.159587e-06   1.528993e-02
##             P4             P5             P6         maxgeno           maxP
##   2.535163e-01   5.074307e-01   2.237600e-01   4.951049e+00   9.780702e-01
##           geno Posterior_Mean
##   5.095652e+00   4.939654e+00
```

```r
# we can always check with str()
```

- Show two ways to extract the third element of `col_means`

```r
# first way
fpout_02$col_means[c(3)]
```

```
##           P1
## 5.571926e-24
```

```r
#second way
fpout_02$col_means[c("P1")]
```

```
##           P1
## 5.571926e-24
```

```r
#third way (optional)
fpout_02$col_means[c(-1, -2, -4, -5, -6, -7, -8, -9, -10, -11, -12)]
```

```
##           P1
## 5.571926e-24
```

For Loops

Consider the recursive sequence defined by

$$x_n = x_{n-1} + \frac{|x_{n-3} - x_{n-2}|}{4}.$$

That is, element n is the sum of element $n-1$ and the absolute value of the difference between between elements $n-3$ and $n-2$ divided by two. For example, if we let $x_1 = 3$, $x_2 = 1$, and $x_3 = 10$, then x_4 is

$$x_4 = 10 + \frac{|3-1|}{4} = 11.$$

1. Write a function called `calcn()` that takes as input a vector `x` containing the first three elements of this sequence and an integer `n` denoting the final element of the sequence to calculate.

- `calcn()` should return element `n`.
- Include error checking to to ensure the inputs are of the correct length and type and n is greater than 0.

For example, in my implementation of `calcn()`, I obtained the following: (see HTML)

- Proper Function structure

```r
calcn <- function(x, n){
  nums <- vector(mode = "integer", length = n) # set nums type
  for(i in seq_along(nums)){
    if (i <= 3){
```

```
      nums[i] <- x[i] # 1, 2, 3 can not be calculated so keep it
    }
    else {
      nums[i] <- nums[i-1] + (abs(nums[i-3] - nums[i-2]))/4 # start working formula with 4
    }

    }
    return(nums[n]) # return Xn in formula
}

calcn(x = c(2, 4, 3), n = 3)
```

```
## [1] 3
```

```
calcn(x = c(2, 4, 3), n = 4)
```

```
## [1] 3.5
```

```
calcn(x = c(2, 4, 3), n = 5)
```

```
## [1] 3.75
```

```
calcn(x = c(2, 4, 3), n = 6)
```

```
## [1] 3.875
```

```
calcn(x = c(2, 4, 3), n = 7)
```

```
## [1] 3.9375
```

- Error checks **Revised: Error checks are not in the right place. Instead of using a complicated If structure with nested elements, use a single if at the top or a stopifnot function**

```
calcn <- function(x, n){
  if(length(x) == 3 & n > 0 & is.integer(n)){
  nums <- vector(mode = "integer", length = n)
  for(i in seq_along(nums)){
    if(i <= 3){
      nums[i] <- x[i]
    }else{
      nums[i] <- nums[i-1] + (abs(nums[i-3] - nums[i-2]))/4
    }

    }
  return(nums[n])
    }else{
      stop("x is not equal to 3 or n is not a positive integer")

    } # error check x and n
} # created calcn function


calcn(x = c(2, 4, 3), n = 7L) # correct x and n type
```

```
## [1] 3.9375
```

```
calcn(x = c(2, 4), n = 7) # length of x < 3
```

```
## Error in calcn(x = c(2, 4), n = 7): x is not equal to 3 or n is not a positive integer
```

```
calcn(x = c(2, 4, 3, 5), n = 6) # length of x > 3
```

```
## Error in calcn(x = c(2, 4, 3, 5), n = 6): x is not equal to 3 or n is not a positive integer
```

```
calcn(x = c(2, 4, 3), n = 0) # n <= 0
```

```
## Error in calcn(x = c(2, 4, 3), n = 0): x is not equal to 3 or n is not a positive integer
```

```
calcn(x = c(2, 4, 3), n = 5.53) # n is a float(dbl)
```

```
## Error in calcn(x = c(2, 4, 3), n = 5.53): x is not equal to 3 or n is not a positive integer
```

- Still not clean. If we want to use stop(), then put the checks at the front and if they are false then stop("error message) and end the if clause.
- Then start the rest of you function. You should not have to change anything about your function code if you wrote it without any error checks and then just adding error checks in the front as separate lines. If you comment them out your function should work.
- Stopifnot is the same only you want to run stopifnot(test is true, test is true)

```
calcn_test <- function(x, n){
  stopifnot(length(x) == 3 & is.numeric(x)) # error check x
  stopifnot(n > 0 & is.integer(n)) # error check n

  nums <- vector(mode = "integer", length = n) # create nums's type
  for(i in seq_along(nums)){
    if(i <= 3){
      nums[i] <- x[i]
    }else{
      nums[i] <- nums[i-1] + (abs(nums[i-3] - nums[i-2]))/4
    }
  }
  return(nums[n])
} # created calcn function


calcn_test(x = c(2, 4, 3), n = 7L) # correct x and n type
```

```
## [1] 3.9375
```

```
calcn_test(x = c(2, 4), n = 7) # length of x < 3
```

```
## Error in calcn_test(x = c(2, 4), n = 7): length(x) == 3 & is.numeric(x) is not TRUE
```

```
calcn_test(x = c(2, 4, 3, 5), n = 6) # length of x > 3
```

```
## Error in calcn_test(x = c(2, 4, 3, 5), n = 6): length(x) == 3 & is.numeric(x) is not TRUE
```

```
calcn_test(x = c(2, 4, 3), n = 0) # n <= 0
```

```
## Error in calcn_test(x = c(2, 4, 3), n = 0): n > 0 & is.integer(n) is not TRUE
```

```
calcn_test(x = c(2, 4, 3), n = 5.53) # n is a float(dbl)
```

```
## Error in calcn_test(x = c(2, 4, 3), n = 5.53): n > 0 & is.integer(n) is not TRUE
```

- Evaluate your function at the following inputs:
  - calcn(c(11,1,130), 1000L)
  - calcn(c(11,1,130), 1L)

- calcn(c(7, 3, 20), 8L)

```
calcn(c(11,1,130), 1000L)
```

```
## [1] 176.3333
```

```
calcn(c(11,1,130), 1L)
```

```
## [1] 11
```

```
calcn(c(7, 3, 20), 8L)
```

```
## [1] 26.625
```

Question: Lists, For-loops, and `map_*()`

Lists are often used to save simulation output. You can then extract individual elements from the lists using for-loops.

Consider the t-test, used to test whether or not the mean of some observations is 0. We would use the following code to simulate data from a Normal (0,1) distribution, and then use a t-test to test if the true mean is 0:

```
x <- rnorm(n = 10, mean = 0, sd = 1)
tout <- t.test(x)
tout
```

```
##
##  One Sample t-test
##
## data:  x
## t = 0.87002, df = 9, p-value = 0.4069
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  -0.4625038  1.0405851
## sample estimates:
## mean of x
## 0.2890407
```

1. The output of `t.test()` is a list-like object. Use one function to show how many elements are in the list along with their names and types.
   Not revise: 3.3.1: - 0.05 No need for additional functions to repeat what is in str()

```
# only one function to show all
str(tout)
```

```
## List of 10
##  $ statistic  : Named num 0.87
##   ..- attr(*, "names")= chr "t"
##  $ parameter  : Named num 9
##   ..- attr(*, "names")= chr "df"
##  $ p.value    : num 0.407
##  $ conf.int   : num [1:2] -0.463 1.041
##   ..- attr(*, "conf.level")= num 0.95
##  $ estimate   : Named num 0.289
##   ..- attr(*, "names")= chr "mean of x"
##  $ null.value : Named num 0
##   ..- attr(*, "names")= chr "mean"
##  $ stderr     : num 0.332
```

```
##  $ alternative: chr "two.sided"
##  $ method     : chr "One Sample t-test"
##  $ data.name  : chr "x"
##  - attr(*, "class")= chr "htest"
```

```
# how many elements
length(tout)
```

```
## [1] 10
```

```
# names
names(tout)
```

```
##  [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
##  [6] "null.value"  "stderr"      "alternative" "method"      "data.name"
```

```
# types
typeof(tout)
```

```
## [1] "list"
```

```
is_double(tout)
```

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

2. Write a for-loop to perform the following three operations on iteration `i`:
    1. Draw 10 random observations from a normal distribution with mean 0 and standard deviation.
    2. Run a t-test on these 10 observations.
    3. Save the output of the t-test as the `i`th element in a list called `tlist`.

- Set the seed to 1 and run for 1000 iterations.
  Revised: 3.3.2: - 0.25 x is unnecessary since there is no need to save results from rnorm. Should create tlist using vector() so you get the length you want and don't constantly have to ask for more memory to add new elements

```
set.seed(1)
tlist <- list()

for(i in 1:1000) {
  x <- rnorm(10, mean = 0, sd = 2)
  tout <- t.test(x)# draw 10 random observations
  tlist[[i]] <- tout# t-test the x for each time then save to the tlist
}

# randomly test some tlists
tlist[[1]]
```

```
##
##  One Sample t-test
##
## data:  x
## t = 0.53557, df = 9, p-value = 0.6052
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  -0.8523895  1.3812007
## sample estimates:
## mean of x
## 0.2644056
```

```
tlist[[20]]
```

```
##
##  One Sample t-test
##
## data:  x
## t = -0.52263, df = 9, p-value = 0.6139
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  -1.5732947  0.9827645
## sample estimates:
##  mean of x
## -0.2952651
```

```
tlist[[300]]
```

```
##
##  One Sample t-test
##
## data:  x
## t = -0.582, df = 9, p-value = 0.5749
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  -2.725734  1.610202
## sample estimates:
## mean of x
## -0.557766
```

3. Use the appropriate map function to extract the sample mean from each test (from 2) and pipe to an appropriate plot to show the sampling distribution of the sample mean.

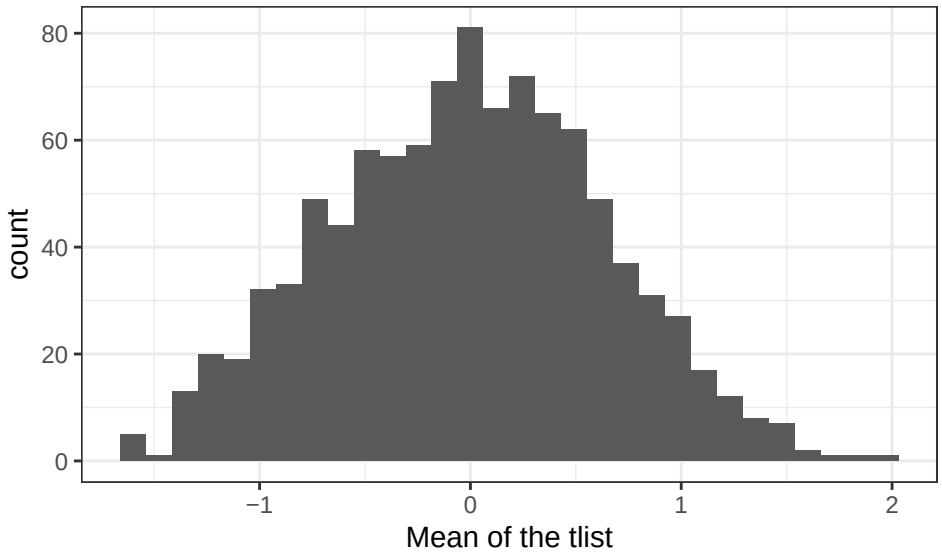
- Hint: Make sure the data going into ggplot is a data frame (tibble)
- QQ plots are for testing against a known distribution or comparing two distributions to see if they are similar.
- Histogram is the more appropriate plot here.
  Revised: 3.3.3: - 0.25 Created intermediate variables instead of using pipe.
  If you don't need intermediate variables they can clutter up your code and memory. Sometimes they can add to clarity or help in debugging but don't create them just because you can. Once you created them and tested everything works consider removing them to have cleaner code.

- Original Data from Yunting

```
tlist %>%
map_dbl(~.$estimate) -> tlist_mean # extract the mean of each tlist
as_tibble(tlist_mean) -> tlist_mean01 # make sure the data save as data frame type

# plot to histogram
tlist_mean01 %>%
 ggplot(aes(x = value)) +
  geom_histogram() +
  theme_bw() +
  xlab("Mean of the tlist")
```

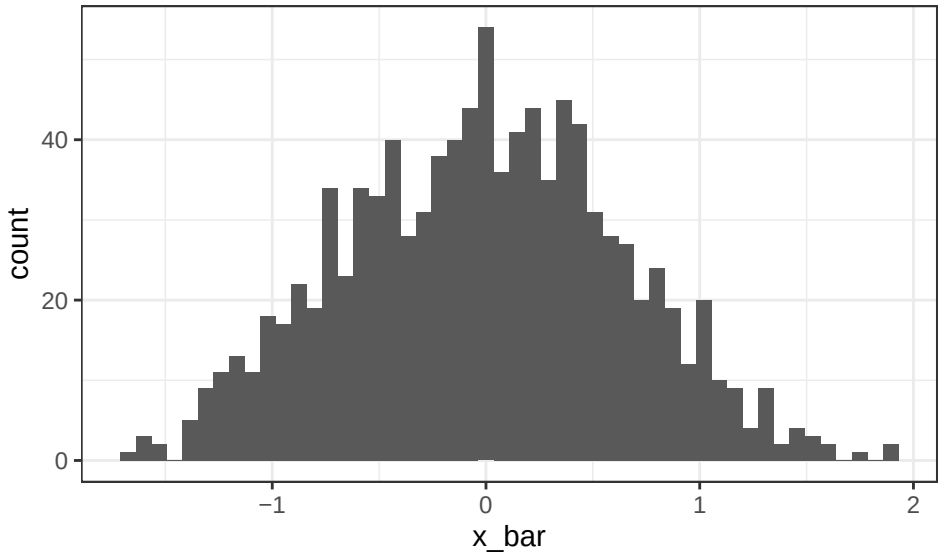
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
#tlist_mean01 %>%
 #ggplot(aes(sample = value))+
  #geom_qq()+
  #theme_bw()
```

- After revised from Dr. Ressler

```
tlist %>%
  map_dbl(~.$estimate[[1]]) %>%
  tibble(x_bar=.) %>%
  ggplot(aes(x=x_bar))+
    geom_histogram(bins = 50)+
  theme_bw()
```



4. The p-value is an important quantity in statistics. Use a for-loop to extract the p-values from each test in part 2 and combine these p-values into a single vector called `pvec_f`. Show the first 6 values. **Revised:3.3.4: - 0.1 Use mode and length = 1000) to initialize vector of proper type and length**

```
pvec_f <- vector(mode = "double", length = 1000) # set pvec_f is a vector type

# start for loop
for(i in 1:1000) {
  pvec_f[[i]] <- tlist[[i]]$p.value # recall tlist$p.value to extract it
}

head(pvec_f)
```

```
## [1] 0.6052327 0.4806056 0.6686761 0.6480420 0.4945143 0.6653161
```

5. Use the appropriate map function to extract the p-values from each test in part 2 and combine these p-values into a single vector called `pvec_m`. Show the first 6 values.

```
map_dbl(tlist, ~.$p.value) -> pvec_m

head(pvec_m)
```

```
## [1] 0.6052327 0.4806056 0.6686761 0.6480420 0.4945143 0.6653161
```

6. *Extra Credit* p-values have a nice property where, if the the null hypothesis is true (i.e. the mean of the observations is actually 0), then the p-values follow the uniform distribution.

- Use the data from `pvec_m` to create a QQ-plot and then interpret the plot with regard to whether the p-values exhibit a uniform distribution.

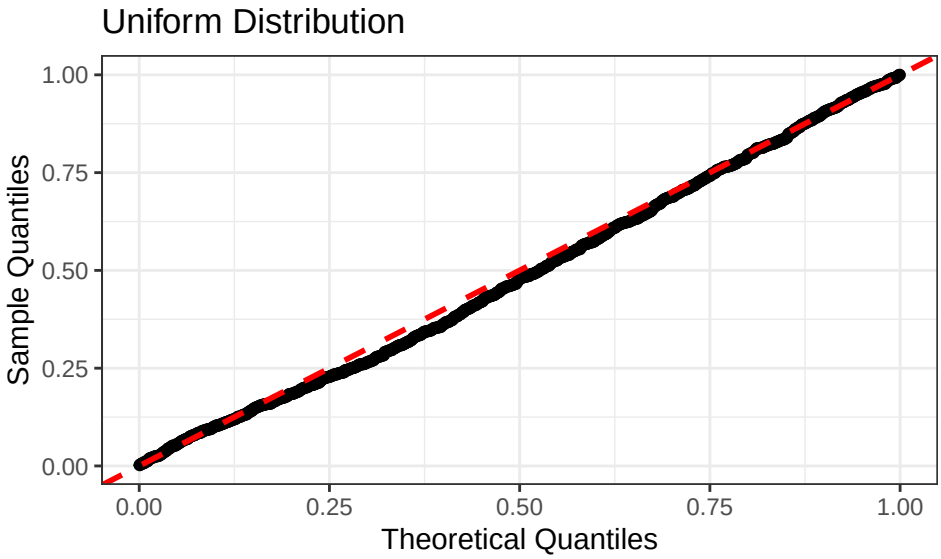
- Ensure your plot has appropriate labels for the axes and a title.
- Include an abline that is dashed and colored red.

Interpretation: If p-values is uniformly distributed, the null hypothesis should be true. In this plot, we can see these two lines are almost sticking together, which means this statistical evidence indicates tlist's p-values is uniformly distributed.
Revised: 3.3.6: +1.75 Labels are confusing as your x axis should not be p values, it is the theoretical uniform distribution and your sample is the p values

```
pvec_m %>%
  as_tibble(pvec_m) -> dfpvec_m # make sure the data save as data frame type
```

```
## Warning: The `validate` argument of `as_tibble()` is deprecated as of tibble 2.0.0.
## Please use the `.name_repair` argument instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_warnings()` to see where this warning was generated.
```

```
  ggplot(data = dfpvec_m)+
  geom_qq(distribution = stats::qunif, mapping = aes(sample = pvec_m))+
  geom_abline(color = "red", linetype = "dashed", size = 1)+ # "qunif" gives the quantile function
  theme_bw()+
  labs(x = "Theoretical Quantiles", y = "Sample Quantiles")+
  ggtitle("Uniform Distribution")
```


Uniform Distribution
Sample Quantiles
Theoretical Quantiles
1.00
0.75
0.50
0.25
0.00
0.00
0.25
0.50
0.75
1.00