## 1. BASIC PROGRAMMING IN R

#### R Program to Find the Factorial of a Number

**Program**

# take input from the user

num = as.integer(readline(prompt="Enter a number: ")) factorial = 1

# check is the number is negative, positive or zero if(num <0) {

print("Sorry, factorial does not exist for negative numbers")

} elseif(num == 0) { print("The factorial of 0 is 1")

} else {

for(i in1:num) { factorial = factorial \* i

}

print(paste("The factorial of", num ,"is",factorial))

}

#### Output:

"The factorial of 6 is 720"

## R Program to check the given number is prime or not

#### Program.

# Program to check if the input number is prime or not # take input from the user

num = as.integer(readline(prompt="Enter a number: ")) flag = 0

# prime numbers are greater than 1 if(num >1) {

# check for factors flag = 1

for(i in2:(num-1)) {

if ((num %% i) == 0) { flag = 0

break

}

}

}

if(num == 2) flag = 1 if(flag == 1) {

print(paste(num,"is a prime number"))

} else {

print(paste(num,"is not a prime number"))

}

#### Output 1

Enter a number: 25

[1] "25 is not a prime number"

#### Output 2

Enter a number: 19

1. "19 is a prime number"

#### R program to perform string functions

**Aim**

To write a R program that performs various string functions, such as concatenation, length calculation, check character is present or not and replace characters

#### Algorithm

* 1. Start
  2. Initialize Strings
  3. Define and assign values to the string variables that will be used in the program
  4. Concatenate two or more strings using paste() function and using separator
  5. Calculate the length of the string by using nchar() function
  6. Check if a character is present in a string or not using grep1() function
  7. Replace a character in the string using gsub() function
  8. Run the R script to perform the string functions and display the output
  9. Stop **Concatenate Strings in R** # create two strings string1 <- "Programiz" string2 <- "Pro"

# using paste() to concatenate two strings result = paste(string1, string2) print(result)

**Output**

"Programiz Pro"

#### Concatenate Strings Using a Separator

# create two strings string1 = "Programiz" string2 = "Pro"

# concatenate two strings using separator result = paste(string1, string2, sep = "-") print(result)

**Output**

Programiz-Pro

**Length of a String in R** # create a string string1<- "Programiz"

# usenchar() to find length of string1 result<- nchar(string1)

cat("Total Length:", result)

#### Output:

Total Length: 9

#### Check if Character is Present in a String in R

string1 <- "Programiz" value1 <- "miz"

# check if"miz" is present in "Programiz" grepl(value1, string1) # TRUE

value2 <- "grm"

# check if"grm" is present in "Programiz" grepl(value2, string1) # FALSE

#### Output:

TRUE FALSE

#### R Program to Replace Characters

string1 <- "Programiz"

# replace "z" with "ng" using gsub() gsub("z", "ng", string1) # programing

#### Output:

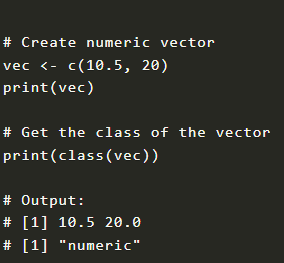
Programing

#### IMPLEMENTATION OF DATATYPES IN R

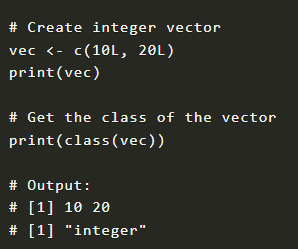
1. **Vector**

A vector is a sequence of data elements of the same basic type.

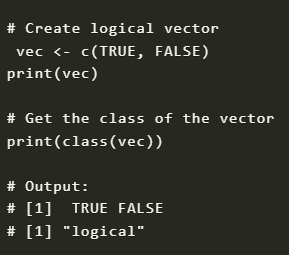
* 1. **Numeric**: It represents real numbers(floating-point)



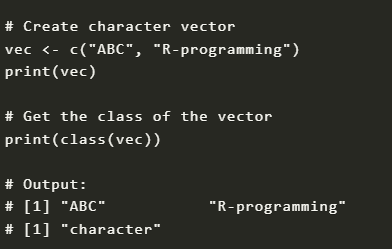
* 1. **Integer**: It represents whole numbers. In this vector, integers have L at the end of itself



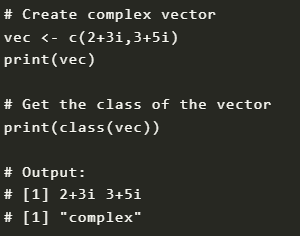
* 1. **Logical**: It represents a boolean value.



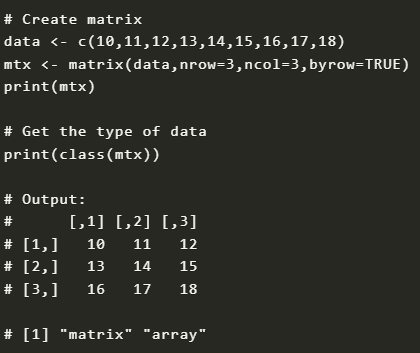
* 1. **Character**: It represents a single/group of characters or strings.



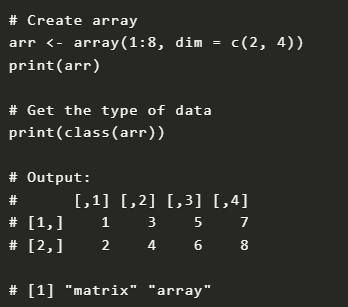
* 1. **Complex:** This represents complex numbers



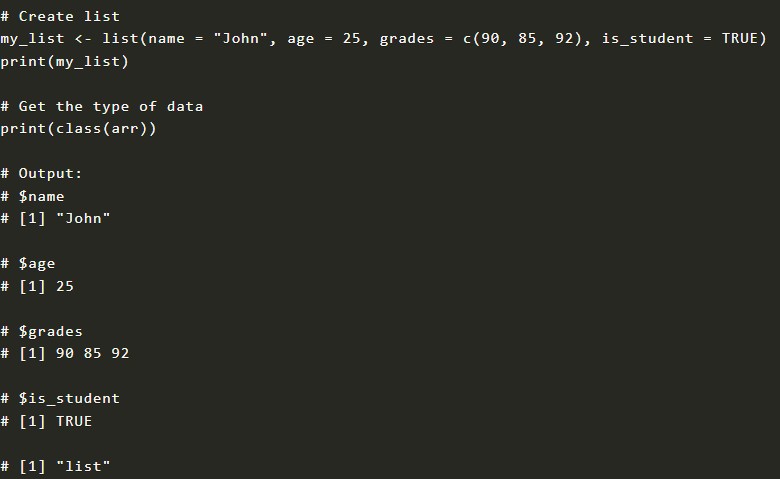
#### Matrix



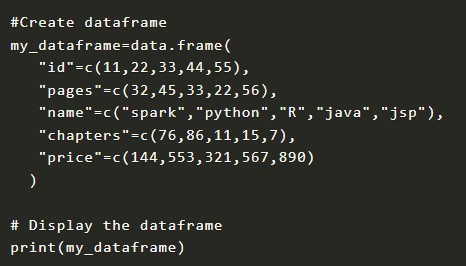
1. **Array**



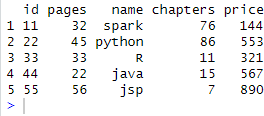
#### List



1. **DataFrame**

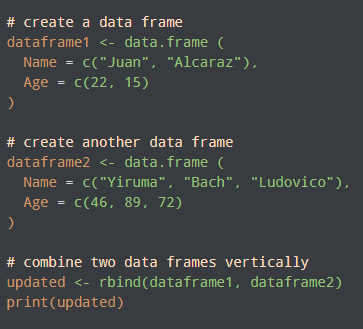


Output:

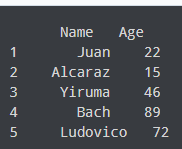


#### R Program to Combine Two Dataframe into One

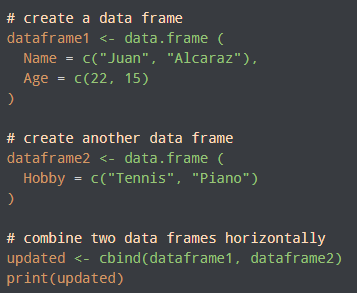
Combine Dataframe Vertically Using rbind() in R



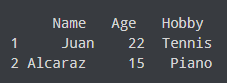
Output



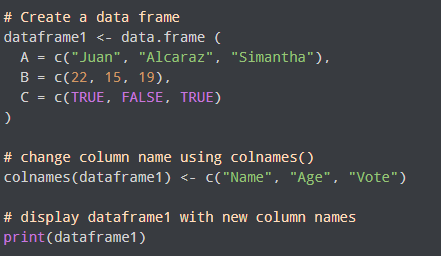
Combine Dataframe Horizontally Using cbind() in R



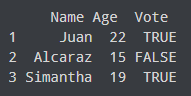
Output



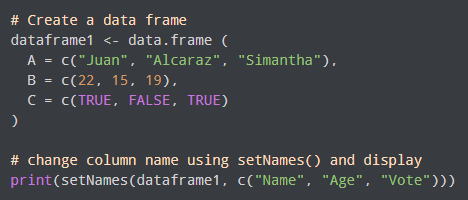
Program to Change Column Name of a Dataframe



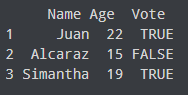
Output:



Change Column Name Using setNames()



Output:



#### RANDOM NUMBER GENERATION

**Program:**

#### Uniform Distribution

>runif(1) # generates 1 random number [1] 0.3984754

>runif(3) # generates 3 random number [1] 0.80902840.17972320.6803607

>runif(3, min=5, max=10) # define the range between 5 and 10 [1] 7.0997818.3554615.173133

# Generate 1000 random numbers from a uniform distribution min\_value<- 0

max\_value<- 50

num\_samples<- 100

random\_numbers<- runif(num\_samples, min = min\_value, max = max\_value)

# Display the random numbers print((random\_numbers))

#### Output

[1] 35.4505621 18.1314228 31.0543197 12.0679666 48.0214440 45.2029749

18.6302422 39.7117077

[9] 3.3255042 30.6856626 29.4262778 47.9781061 6.4419499 33.3063414 9.1940591

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 33.1732024 |  | | | |
| [17] 44.7446287 16.2897554 | 21.5510468 | 27.9254874 | 37.9607891 | 19.9473376 |
| 6.5201459 28.1441231 |  |  |  |  |
| [25] 41.2620238 28.9692573 | 33.3824824 | 48.4199838 | 36.9222070 | 34.2833122 |
| 5.6951293 8.3539030 |  |  |  |  |
| [33] 15.2731601 44.9435134 | 38.4870510 | 26.0111747 | 5.7985457 | 16.1354679 |
| 9.1729476 18.2268161 |  |  |  |  |
| [41] 28.7909359 6.1848184 | 28.0469728 | 26.6758265 | 22.7873698 | 7.6540485 |
| 48.1072010 9.5765814 |  |  |  |  |
| [49] 39.6108082 27.2287823 | 13.8042515 | 15.5892856 | 31.8854508 | 17.9979511 |
| 16.6054464 5.5504175 |  |  |  |  |
| [57] 0.2209532 45.3856475 | 24.1899348 | 30.9788498 | 42.8045200 | 18.9646124 |
| 6.9438106 24.1580314 |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [65] 37.1569004 16.1775712 | 40.8514292 | 24.8639324 | 39.8682737 | 18.1273536 |
| 13.9172852 23.0149862 |  |  |  |  |
| [73] 30.1216386 16.2656533 | 39.0605919 | 13.5853447 | 32.3000609 | 42.8318474 |
| 34.4325746 17.3893671 |  |  |  |  |
| [81] 2.5460618 27.8254792 | 8.1128624 | 22.0050954 | 6.4964783 | 29.4707365 |

25.4657186 48.3868527

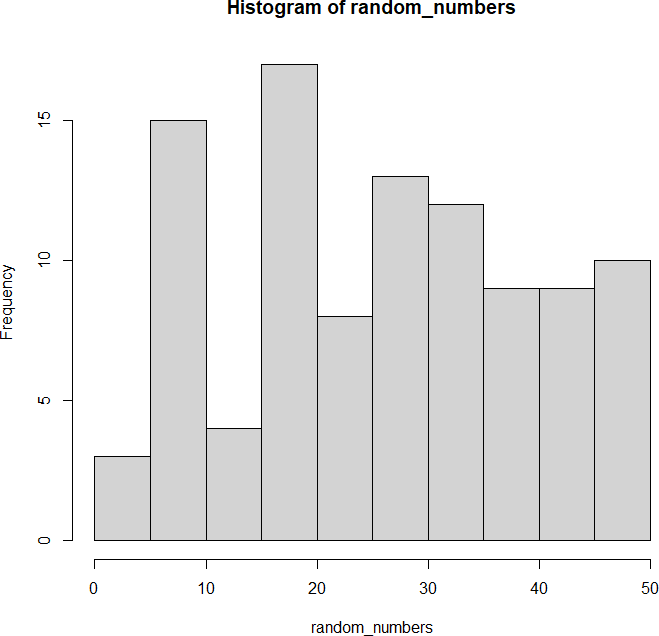
[89] 6.2036748 20.1668791 46.8615398 42.0062728 26.9370341 48.5755289

45.8029313 43.3955000

[97] 16.9225972 40.8434304 33.6589191 18.8792189

#To display the histogram

hist(random\_numbers)



#To compute the mean and standard deviations mean(random\_numbers)

[1] 25.96492

sd(random\_numbers) [1] 13.51788

#### Normal Distribution (Gaussian Distribution)

>rnorm(1) # generates 1 random number [1] 1.072712

>rnorm(3) # generates 3 random number [1] -1.13836560.2016713-0.4602043

>rnorm(3, mean=10, sd=2) # provide own mean and standard deviation [1] 9.8569339.02428610.822507

# Create a vector of 1000 random numbers # with mean=90 and sd=5

x <- rnorm(10000, mean=90, sd=5) # Create the histogram with 50 bars hist(x, breaks=50)

#### Output:

[1] 40.8043964 11.0927212 24.0105955 11.2627842 2.4685370 31.5675786 43.7663135 35.1041564

[9] 3.3868480 34.6328387 2.5157689 13.7591692 1.1443371 20.5695780 26.0985473 8.7174401

[17] 15.9303576 42.4249645 5.0909482 48.0409796 32.8460760 27.4778909 29.2692752 3.9245

[25] 36.7706295 11.2520571 0.2335717 26.6344203 13.9971538 49.2098102 44.7250208 38.76980

[33] 3.4445337 31.8784399 15.5091192 26.0065765 48.3512902 4.9423666 34.3756388 20.33909

[41] 28.4745099 22.1891297 18.9303234 42.7187521 7.4502454 20.5874925 38.8177979 11.17207

[49] 21.8836261 44.8998495 44.5362053 27.4186524 30.7468439 32.5623768 6.6915281 0.7145247

[57] 48.6793808 20.4640119 21.6198104 43.9599116 28.3202471 33.9488599 28.0900611 45.17438

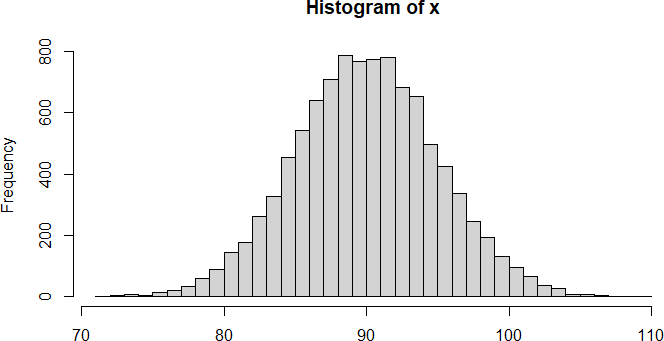
[65] 3.9259795 44.1965642 30.8038562 45.3228294 27.8352947 6.3550454 40.1269306 14.339962

[73] 11.1865474 32.8135226 8.6241119 1.3069708 36.5804295 13.5437287 16.5188219 46.669844

[81] 24.6899221 25.9259868 7.9972449 31.3586005 21.8788709 0.6695888 7.5767456 28.310038

[89] 27.9482520 17.7149408 41.0771061 39.2952338 1.0900171 17.1972097 24.7697607 4.294904

[97] 20.8227233 15.5335120 4.6582193 14.0992036



dnorm:

dnorm() function in R programming measures density function of distribution. # creating a sequence of values

# between -15 to 15 with a difference of 0.1 x = seq(-15, 15, by=0.1)

y = dnorm(x, mean(x), sd(x)) print(y)

# Plot the graph. plot(x, y)

#### Output:

[1] 0.01038093 0.01058784 0.01079745 0.01100975 0.01122475 0.01144243 0.01166279 0.0118858

3

[9] 0.01211154 0.01233990 0.01257091 0.01280456 0.01304082 0.01327969 0.01352115 0.013765

19

[17] 0.01401178 0.01426090 0.01451254 0.01476666 0.01502326 0.01528229 0.01554374 0.015807

58

[25] 0.01607377 0.01634229 0.01661310 0.01688616 0.01716146 0.01743893 0.01771855 0.018000

29

[33] 0.01828408 0.01856990 0.01885770 0.01914743 0.01943904 0.01973250 0.02002773 0.020324

70

[41] 0.02062336 0.02092363 0.02122548 0.02152884 0.02183366 0.02213986 0.02244740 0.022756

21

[49] 0.02306621 0.02337736 0.02368957 0.02400279 0.02431694 0.02463194 0.02494774 0.025264

24

[57] 0.02558139 0.02589909 0.02621729 0.02653588 0.02685481 0.02717398 0.02749331 0.027812

72

[65] 0.02813213 0.02845146 0.02877060 0.02908949 0.02940803 0.02972614 0.03004372 0.030360

68

[73] 0.03067694 0.03099240 0.03130697 0.03162056 0.03193308 0.03224442 0.03255451 0.032863

24

[81] 0.03317051 0.03347624 0.03378033 0.03408268 0.03438320 0.03468179 0.03497835 0.035272

80

[89] 0.03556503 0.03585494 0.03614245 0.03642745 0.03670985 0.03698956 0.03726648 0.037540

52

[97] 0.03781158 0.03807958 0.03834440 0.03860598 0.03886421 0.03911900 0.03937026 0.039617

90

[105] 0.03986185 0.04010200 0.04033827 0.04057057 0.04079883 0.04102296 0.04124287 0.04145

849

[113] 0.04166974 0.04187653 0.04207879 0.04227645 0.04246943 0.04265766 0.04284107 0.04301

959

[121] 0.04319315 0.04336168 0.04352513 0.04368343 0.04383651 0.04398432 0.04412681 0.04426

392

[129] 0.04439559 0.04452177 0.04464242 0.04475748 0.04486692 0.04497069 0.04506875 0.04516

107

[137] 0.04524759 0.04532830 0.04540316 0.04547214 0.04553522 0.04559236 0.04564354 0.04568

876

[145] 0.04572798 0.04576119 0.04578838 0.04580955 0.04582467 0.04583374 0.04583677 0.04583

374

[153] 0.04582467 0.04580955 0.04578838 0.04576119 0.04572798 0.04568876 0.04564354 0.04559

236

[161] 0.04553522 0.04547214 0.04540316 0.04532830 0.04524759 0.04516107 0.04506875 0.04497

069

[169] 0.04486692 0.04475748 0.04464242 0.04452177 0.04439559 0.04426392 0.04412681 0.04398

432

[177] 0.04383651 0.04368343 0.04352513 0.04336168 0.04319315 0.04301959 0.04284107 0.04265

766

[185] 0.04246943 0.04227645 0.04207879 0.04187653 0.04166974 0.04145849 0.04124287 0.04102

296

[193] 0.04079883 0.04057057 0.04033827 0.04010200 0.03986185 0.03961790 0.03937026 0.03911

900

[201] 0.03886421 0.03860598 0.03834440 0.03807958 0.03781158 0.03754052 0.03726648 0.03698

956

[209] 0.03670985 0.03642745 0.03614245 0.03585494 0.03556503 0.03527280 0.03497835 0.03468

179

[217] 0.03438320 0.03408268 0.03378033 0.03347624 0.03317051 0.03286324 0.03255451 0.03224

442

[225] 0.03193308 0.03162056 0.03130697 0.03099240 0.03067694 0.03036068 0.03004372 0.02972

614

[233] 0.02940803 0.02908949 0.02877060 0.02845146 0.02813213 0.02781272 0.02749331 0.02717

398

[241] 0.02685481 0.02653588 0.02621729 0.02589909 0.02558139 0.02526424 0.02494774 0.02463

194

[249] 0.02431694 0.02400279 0.02368957 0.02337736 0.02306621 0.02275621 0.02244740 0.02213

986

[257] 0.02183366 0.02152884 0.02122548 0.02092363 0.02062336 0.02032470 0.02002773 0.01973

250

[265] 0.01943904 0.01914743 0.01885770 0.01856990 0.01828408 0.01800029 0.01771855 0.01743

893

[273] 0.01716146 0.01688616 0.01661310 0.01634229 0.01607377 0.01580758 0.01554374 0.01528

229

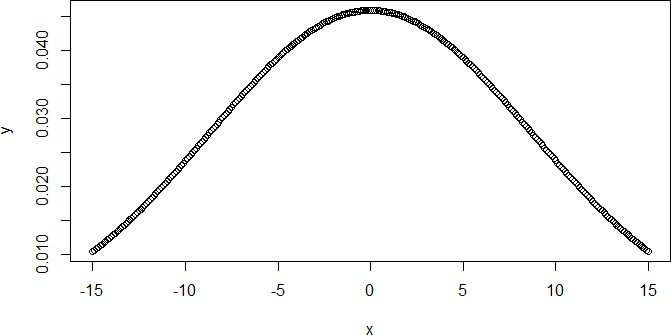
[281] 0.01502326 0.01476666 0.01451254 0.01426090 0.01401178 0.01376519 0.01352115 0.01327

969

[289] 0.01304082 0.01280456 0.01257091 0.01233990 0.01211154 0.01188583 0.01166279 0.01144

243

[297] 0.01122475 0.01100975 0.01079745 0.01058784 0.01038093



#### pnorm()

pnorm() function is the cumulative distribution function which measures the probability that a random number X takes a value less than or equal to x.

[1] 2.052263e-10 2.823158e-10 3.874147e-10 5.303423e-10 7.242292e-10 9.865876e-10 1.340712e- 09

[8] 1.817508e-09 2.457865e-09 3.315746e-09 4.462172e-09 5.990371e-09 8.022392e-09 1.071759e- 08

[15] 1.428348e-08 1.898956e-08 2.518491e-08 3.332045e-08 4.397712e-08 5.790134e-08 7.604961e-08

[22] 9.964426e-08 1.302432e-07 1.698267e-07 2.209050e-07 2.866516e-07 3.710674e-07 4.791833e-07

[29] 6.173074e-07 7.933282e-07 1.017083e-06 1.300807e-06 1.659675e-06 2.112455e-06 2.682296e-06

[36] 3.397673e-06 4.293514e-06 5.412544e-06 6.806877e-06 8.539905e-06 1.068853e-05 1.334575e-05

[43] 1.662376e-05 2.065751e-05 2.560882e-05 3.167124e-05 3.907560e-05 4.809634e-05 5.905891e-05

[50] 7.234804e-05 8.841729e-05 1.077997e-04 1.311202e-04 1.591086e-04 1.926156e-04 2.326291e-04

[57] 2.802933e-04 3.369293e-04 4.040578e-04 4.834241e-04 5.770250e-04 6.871379e-04 8.163523e-04

[64] 9.676032e-04 1.144207e-03 1.349898e-03 1.588870e-03 1.865813e-03 2.185961e-03 2.555130e-03

[71] 2.979763e-03 3.466974e-03 4.024589e-03 4.661188e-03 5.386146e-03 6.209665e-03 7.142811e-03

[78] 8.197536e-03 9.386706e-03 1.072411e-02 1.222447e-02 1.390345e-02 1.577761e-02 1.786442e-02

[85] 2.018222e-02 2.275013e-02 2.558806e-02 2.871656e-02 3.215677e-02 3.593032e-02 4.005916e-02

[92] 4.456546e-02 4.947147e-02 5.479929e-02 6.057076e-02 6.680720e-02 7.352926e-02 8.075666e-02

[99] 8.850799e-02 9.680048e-02 1.056498e-01 1.150697e-01 1.250719e-01 1.356661e-01 1.468591e-01

[106] 1.586553e-01 1.710561e-01 1.840601e-01 1.976625e-01 2.118554e-01 2.266274e-01 2.419637e-01

[113] 2.578461e-01 2.742531e-01 2.911597e-01 3.085375e-01 3.263552e-01 3.445783e-01 3.631693e-01

[120] 3.820886e-01 4.012937e-01 4.207403e-01 4.403823e-01 4.601722e-01 4.800612e-01 5.000000e-01

[127] 5.199388e-01 5.398278e-01 5.596177e-01 5.792597e-01 5.987063e-01 6.179114e-01 6.368307e-01

[134] 6.554217e-01 6.736448e-01 6.914625e-01 7.088403e-01 7.257469e-01 7.421539e-01 7.580363e-01

[141] 7.733726e-01 7.881446e-01 8.023375e-01 8.159399e-01 8.289439e-01 8.413447e-01 8.531409e-01

[148] 8.643339e-01 8.749281e-01 8.849303e-01 8.943502e-01 9.031995e-01 9.114920e-01 9.192433e-01

[155] 9.264707e-01 9.331928e-01 9.394292e-01 9.452007e-01 9.505285e-01 9.554345e-01 9.599408e-01

[162] 9.640697e-01 9.678432e-01 9.712834e-01 9.744119e-01 9.772499e-01 9.798178e-01 9.821356e-01

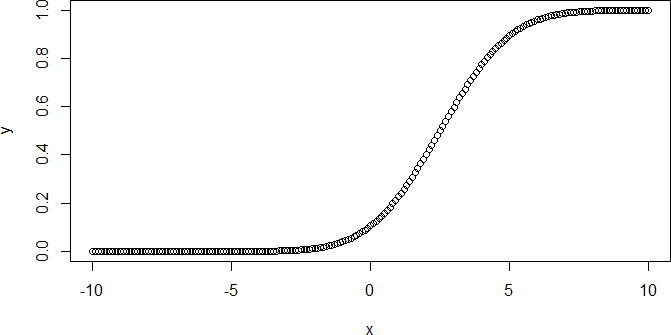
[169] 9.842224e-01 9.860966e-01 9.877755e-01 9.892759e-01 9.906133e-01 9.918025e-01 9.928572e-01

[176] 9.937903e-01 9.946139e-01 9.953388e-01 9.959754e-01 9.965330e-01 9.970202e-01 9.974449e-01

[183] 9.978140e-01 9.981342e-01 9.984111e-01 9.986501e-01 9.988558e-01 9.990324e-01 9.991836e-01

[190] 9.993129e-01 9.994230e-01 9.995166e-01 9.995959e-01 9.996631e-01 9.997197e-01 9.997674e-01

[197] 9.998074e-01 9.998409e-01 9.998689e-01 9.998922e-01 9.999116e-01



#### qnorm()

qnorm() function is the inverse of pnorm() function. It takes the probability value and gives output which corresponds to the probability value. It is useful in finding the percentiles of a normal distribution.

**PROGRAM:**

# Create a sequence of probability values # incrementing by 0.02.

x <- seq(0, 1, by = 0.02)

y <- qnorm(x, mean(x), sd(x)) print(y)

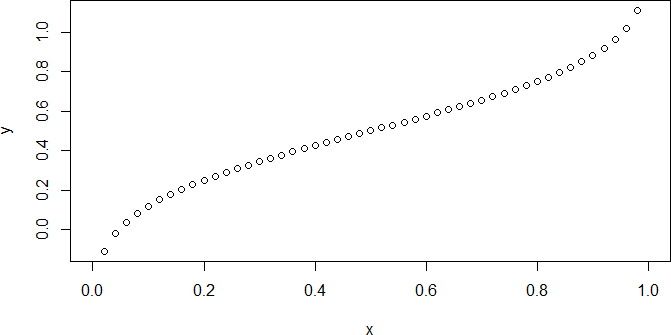
# Plot the graph. plot(x, y)

-Inf -0.11062345 -0.02051639 0.03773258 0.08224219 0.11896733 0.15065131 0.17879797

[9] 0.20432641 0.22784239 0.24976802 0.27041045 0.29000115 0.30871966 0.32670876 0.344

08452

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| [17] 0.36094315 | 0.37736590 | 0.39342254 | 0.40917403 | 0.42467449 | 0.43997275 | 0.45511363 | 0.47 |
| 013891 |  |  |  |  |  |  |  |
| [25] 0.48508827 | 0.50000000 | 0.51491173 | 0.52986109 | 0.54488637 | 0.56002725 | 0.57532551 | 0.59 |
| 082597 |  |  |  |  |  |  |  |
| [33] 0.60657746 | 0.62263410 | 0.63905685 | 0.65591548 | 0.67329124 | 0.69128034 | 0.70999885 | 0.72 |
| 958955 |  |  |  |  |  |  |  |
| [41] 0.75023198 | 0.77215761 | 0.79567359 | 0.82120203 | 0.84934869 | 0.88103267 | 0.91775781 | 0.96 |
| 226742 |  |  |  |  |  |  |  |
| [49] 1.02051639 | 1.11062345 | Inf |  |  |  |  |  |



1. Exponential Distribution dexp() Function

The dexp() function returns the corresponding values of the exponential density for an input vector of

quantiles.

#### Syntax:

dexp(x\_dexp, rate)

**PROGRAM:**

# R program to illustrate # exponential distribution # Specify x-values

x\_dexp<- seq(1, 10, by = 0.1) # Apply dexp() function

y\_dexp<- dexp(x\_dexp, rate = 5) print (y\_dexp)

# Plot dexp values plot(y\_dexp)

[1] 3.368973e-02 2.043386e-02 1.239376e-02 7.517196e-03 4.559410e-03 2.765422e-03 1.677313e- 03

[8] 1.017342e-03 6.170490e-04 3.742591e-04 2.269996e-04 1.376822e-04 8.350850e-05 5.065047e- 05

[15] 3.072106e-05 1.863327e-05 1.130165e-05 6.854795e-06 4.157644e-06 2.521738e-06 1.529512e

-06

[22] 9.276957e-07 5.626759e-07 3.412802e-07 2.069969e-07 1.255500e-07 7.614990e-08 4.618725e

-08

[29] 2.801398e-08 1.699134e-08 1.030577e-08 6.250764e-09 3.791280e-09 2.299528e-09 1.394734e

-09

[36] 8.459490e-10 5.130940e-10 3.112072e-10 1.887567e-10 1.144867e-10 6.943972e-11 4.211732e

-11

[43] 2.554545e-11 1.549410e-11 9.397644e-12 5.699959e-12 3.457200e-12 2.096898e-12 1.271833e

-12

[50] 7.714056e-13 4.678811e-13 2.837843e-13 1.721239e-13 1.043984e-13 6.332083e-14 3.840602e

-14

[57] 2.329443e-14 1.412879e-14 8.569542e-15 5.197690e-15 3.152558e-15 1.912123e-15 1.159761e

-15

[64] 7.034309e-16 4.266524e-16 2.587778e-16 1.569566e-16 9.519901e-17 5.774112e-17 3.502176e

-17

[71] 2.124177e-17 1.288379e-17 7.814411e-18 4.739680e-18 2.874761e-18 1.743631e-18 1.057566e

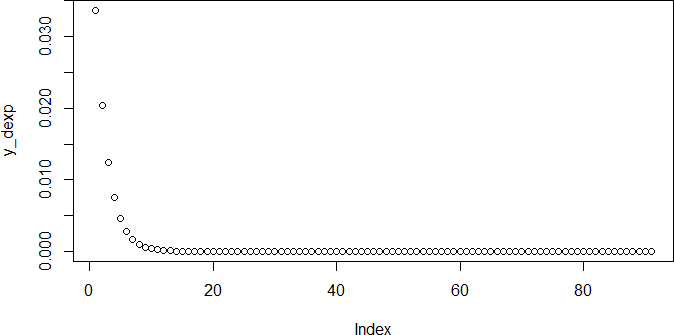
-18

[78] 6.414459e-19 3.890566e-19 2.359748e-19 1.431259e-19 8.681026e-20 5.265309e-20 3.193571e

-20

[85] 1.936999e-20 1.174849e-20 7.125820e-21 4.322029e-21 2.621443e-21 1.589985e-21 9.643749e

-22



#### pexp() Function

The **pexp()** function returns the corresponding values of the exponential [cumulative distribution](https://www.geeksforgeeks.org/plot-cumulative-distribution-function-in-r/) [function](https://www.geeksforgeeks.org/plot-cumulative-distribution-function-in-r/) for an input vector of quantiles.

#### Syntax:

pexp(x\_pexp, rate )

# R program to illustrate # exponential distribution # Specify x-values

x\_pexp<- seq(1, 10, by = 0.2) # Apply pexp() function

y\_pexp<- pexp(x\_pexp, rate = 1) print(y\_pexp)

# Plot values plot(y\_pexp)

#### Output :

[1] 0.6321206 0.6988058 0.7534030 0.7981035 0.8347011 0.8646647 0.8891968 0.9092820 0.92572

64

[10] 0.9391899 0.9502129 0.9592378 0.9666267 0.9726763 0.9776292 0.9816844 0.9850044 0.9877

227

[19] 0.9899482 0.9917703 0.9932621 0.9944834 0.9954834 0.9963021 0.9969724 0.9975212 0.9979

706

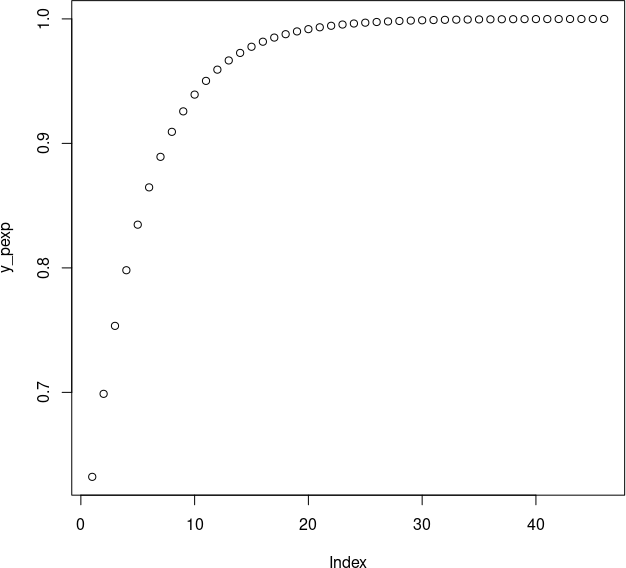
[28] 0.9983384 0.9986396 0.9988862 0.9990881 0.9992534 0.9993887 0.9994995 0.9995903 0.9996

645

[37] 0.9997253 0.9997751 0.9998159 0.9998493 0.9998766 0.9998990 0.9999173 0.9999323 0.9999

445

[46] 0.9999546



*Cumulative Exponential Distribution Function*

#### qexp() Function

The **qexp()** function gives the possibility, we can use the qexp function to return the corresponding values of the quantile function.

Syntax: qexp(x\_qexp, rate)

**PROGRAM:**

# R program to illustrate # exponential distribution # Specify x-values

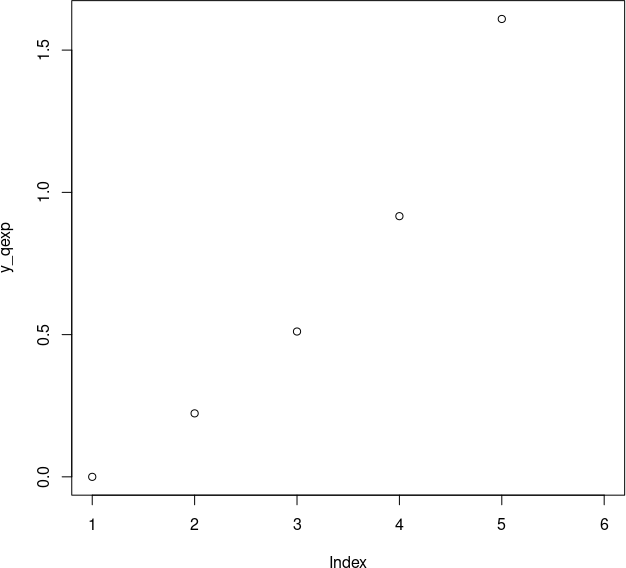
x\_qexp<- seq(0, 1, by = 0.2)

# Apply qexp() function y\_qexp<- qexp(x\_qexp, rate = 1) print(y\_qexp)

# Plot values plot(y\_qexp)

#### Output:

[1] 0.0000000 0.2231436 0.5108256 0.9162907 1.6094379 Inf



*Quantile Function of Exponential Distribution*

#### rexp() Function

The **rexp()** function is used to simulate a set of random numbers drawn from the exponential distribution.

#### Syntax:

rexp(N, rate )

**PROGRAM:**

# R program to illustrate # exponential distribution

# Set seed for reproducibility set.seed(500)

# Specify size N <- 100

# Draw exp distributed values y\_rexp<- rexp(N, rate = 1) print(y\_rexp)

# Plot exp density

hist(y\_rexp, breaks = 50, main = "")

#### Output:

[1] 0.66719993 0.45002358 0.14257749 0.02436381 0.49322762 0.82139088 0.11391668

0.46414880

[9] 0.70557190 0.01675899 0.23562757 0.21656866 0.70681793 2.32981641 0.61863613

2.97037599

[17] 0.07975247 0.76414325 0.62810028 0.38892964 0.02759131 2.15895799 3.00698305

0.23519352

[25] 2.65089996 1.19958381 0.68556176 0.65079256 3.08099527 0.03189506 1.37927999

0.66687517

[33] 1.09230909 0.85757059 1.71037590 2.18633150 1.92846409 0.30167662 2.35071957

0.62453294

[41] 0.01865751 0.54126454 0.68868353 1.23938937 0.02341995 1.50643960 1.04310060

2.41012879

[49] 2.78019968 1.91069105 0.97774924 3.56169077 1.06635904 0.16756170 1.24567482

0.37094422

[57] 0.14898650 0.32631005 0.21306997 0.60338641 1.03829997 0.94280470 1.49923479

3.10367180

[65] 0.91364583 2.38616162 0.07679897 2.69974705 0.75232513 1.36956192 0.06836016

0.13301195

[73] 0.70949794 0.60779366 0.72738493 0.29846874 0.08165857 1.34937556 0.49909595

1.32655869

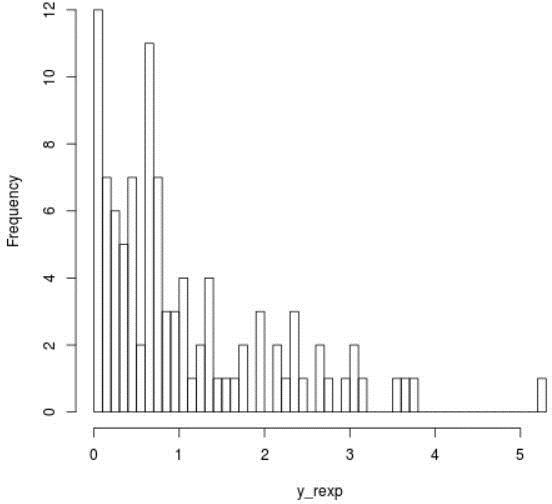
[81] 3.72322864 0.44414568 0.06709964 0.10721554 0.04384922 2.26735044 0.40306880

0.78559992

[89] 0.36241435 0.89872529 0.28555096 0.61031540 0.47167679 3.60849667 1.96123075

1.60011004

[97] 0.59441929 0.15330044 5.23070219 1.79770822



*Histogram of 100 Exponentially Distributed Numbers*

#### Binomial Distribution in R Programming

**Functions for Binomial Distribution**

dbinom()

pbinom()

dbinom(k, n, p)

pbinom(k, n, p)

where n is total number of trials, p is probability of success, k is the value at which the probability has to be found out.

qbinom()

qbinom(P, n, p)

Where P is the probability, n is the total number of trials and p is the probability of success.

rbinom()

rbinom(n, N, p)

Where n is numbers of observations, N is the total number of trials, p is the probability of success.

#### dbinom() Function

dbinom(3, size = 13, prob = 1 / 6)

probabilities <- dbinom(x = c(0:10), size = 10, prob = 1 / 6) data.frame(x, probs)

plot(0:10, probabilities, type = "l")

#### Output :

>dbinom(3, size = 13, prob = 1/6) [1] 0.2138454

>probabilities = dbinom(x = c(0:10), size = 10, prob = 1/6)

>data.frame(probabilities) probabilities

1 1.615056e-01

2 3.230112e-01

3 2.907100e-01

4 1.550454e-01

5 5.426588e-02

6 1.302381e-02

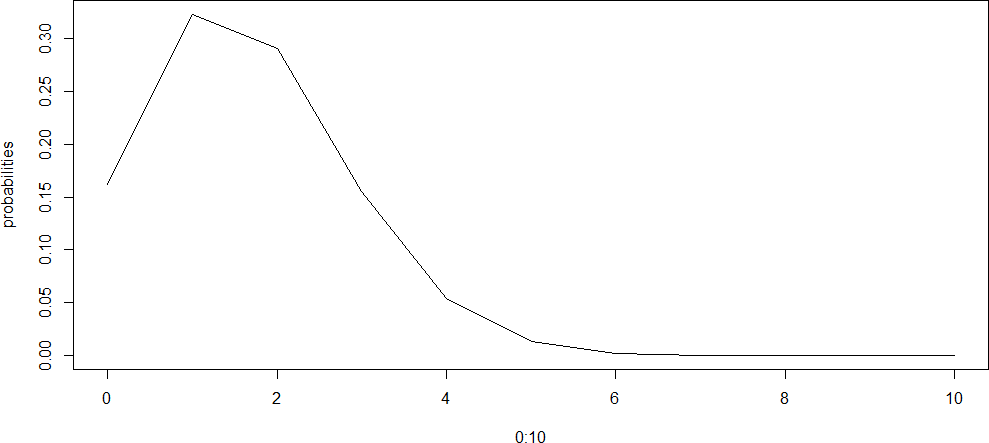
7 2.170635e-03

8 2.480726e-04

9 1.860544e-05

10 8.269086e-07

11 1.653817e-08

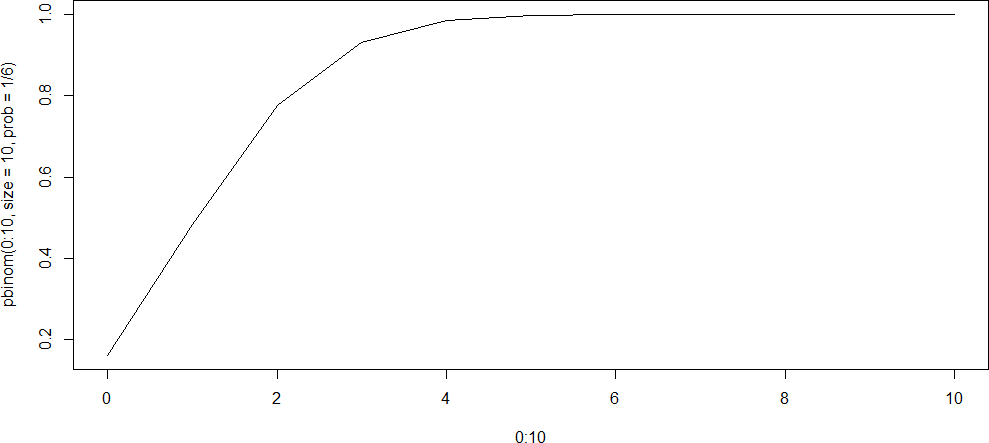


#### pbinom() Function

t(0:10, pbinom(0:10, size = 10, prob = 1 / 6), type = "l")

#### Output :

>pbinom(3, size = 13, prob = 1/6) [1] 0.8419226



*qbinom() Function*

This function is used to find the nth quantile, that is if P(x <= k) is given, it finds k.

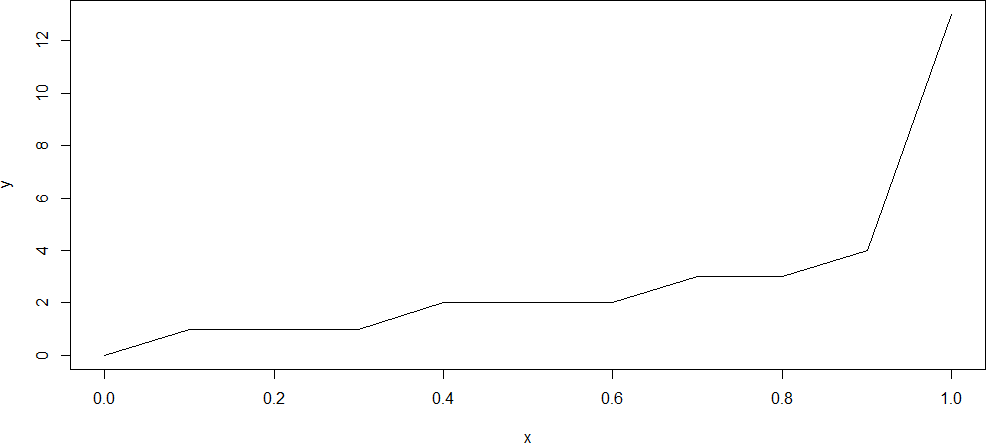
qbinom(0.8419226, size = 13, prob = 1 / 6) x <- seq(0, 1, by = 0.1)

y <- qbinom(x, size = 13, prob = 1 / 6) plot(x, y, type = 'l')

#### Output :

>qbinom(0.8419226, size = 13, prob = 1/6)

[1] 3



***rbinom() Function***

This function generates n random variables of a particular probability.

#### Syntax:

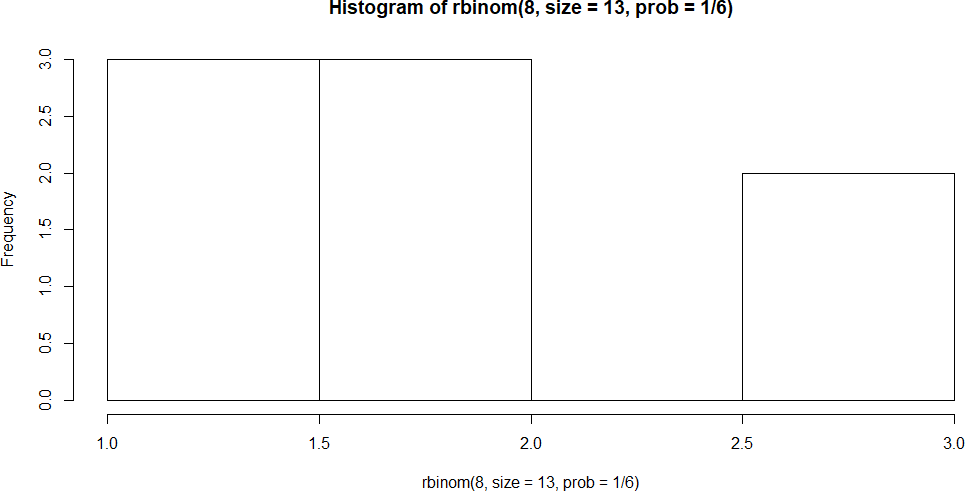
rbinom(n, N, p)

#### Example:

rbinom(8, size =13, prob =1/6) hist(rbinom(8, size =13, prob =1/6))

#### Output:

>rbinom(8, size = 13, prob = 1/6) [1] 1 1 2 1 4 0 2 3



#### Poisson Distribution in R Programming

* The Poisson distribution represents the probability of a provided number of cases happening in a set period of space or time if these cases happen with an identified constant mean rate (free of the period since the ultimate event).
* Poisson distribution has been named after Siméon Denis Poisson (French Mathematician).
* There are four Poisson functions available in R:
  + dpois
  + ppois
  + qpois
  + rpois

#### Poisson Probability Mass Function – dpois()

This function is used for the illustration of Poisson density in an R plot. The function dpois() calculates the probability of a random variable that is available within a certain range.

##### Syntax:

*dpois(k,λ,log)*

##### where,

* ***K:*** *number of successful events happened in an interval*
* ***lambda****: mean per interval*
* ***log:*** *If TRUE then the function returns probability in form of log*

dpois(2, 3)

dpois(6, 6)

#### Output:

[1] 0.2240418

[1] 0.1606231

#### Poisson Distribution – ppois()

This function is used for the illustration of the cumulative probability function in an R plot. The function ppois() calculates the probability of a random variable that will be equal to or less than a number.

##### Syntax:

*ppois(q,λ,lower.tail,log)*

##### where,

* ***K:*** *number of successful events happened in an interval*
* ***lambda****: mean per interval*
* ***lower.tail:*** *If TRUE then left tail is considered otherwise if the FALSE right tail is considered*
* ***log:*** *If TRUE then the function returns probability in form of log*

ppois(2, 3)

ppois(6, 6)

#### Output:

[1] 0.4231901

[1] 0.6063028

#### Poisson pseudorandom – rpois()

The function rpois() is used for generating random numbers from a given Poisson distribution.

##### Syntax:

*rpois(q,λ)*

##### where,

* ***q:*** *number of random numbers needed*
* ***lambda****: mean per interval*

rpois(2, 3)

rpois(6, 6)

#### Output:

[1] 2 3

[1] 6 7 6 10 9 4

#### Poisson Quantile Function – qpois()

The function qpois() is used for generating the quantile of a given Poisson’s distribution. In probability, quantiles are marked points that divide the graph of a probability distribution into intervals (continuous ) which have equal probabilities.

##### Syntax:

*qpois(q,λ,lower.tail,log)*

##### where,

* ***K:*** *number of successful events happened in an interval*
* ***lambda****: mean per interval*
* ***lower.tail:*** *If TRUE then left tail is considered otherwise if the FALSE right tail is considered*
* ***log:*** *If TRUE then the function returns probability in form of log*

y <- c(.01, .05, .1, .2)

qpois(y, 2)

qpois(y, 6)

#### Output:

[1] 0 0 0 1

[1] 1 2 3 4

#### SAMPLING IN R Sampling with Replacement:

In sampling with replacement, each selected item is returned to the population before the next item is drawn. In R, this behavior is specified using the replace argument in the sample() function.

#sample range lies between 1 to 5 x<- sample(1:5)

#prints the samples x

**Output** ->32154

#samples range is 1 to 5 and number of samples is 3 x<- sample(1:5, 3)

#prints the samples (3 samples) x

**Output** ->245

#specifing replace=TRUE or T will allow repetition of values so that the function will generate 6 samples in the range 1 to 5. Here 2 is repeated.

x<- sample(1:5, 6, replace=T)

**Output ->**242243

#### Sampling from a Vector:

#Create a vector of data population\_vector<- c(10, 20, 30, 40, 50) #Sample 3 values with replacement

sampled\_vector<- sample (population\_vector, size = 3, replace = TRUE) print (sampled\_vector)

Output 50 20 50

#### Sampling from a Data Frame:

# Create a data frame population\_df<- data.frame(

Name = c("Alice", "Bob", "Charlie", "David", "Eve"), Age = c(25, 30, 35, 40, 45)

)

# Sample 2 rows with replacement

sampled\_df<- population\_df[sample(nrow(population\_df), size = 2, replace = TRUE), ] print(sampled\_df)

#### Output

Name Age Bob 30

Charlie 35

#### Sampling from a List:

# Create a list population\_list<- list(

fruits = c("Apple", "Banana", "Cherry", "Date"),

colors = c("Red", "Yellow", "Red", "Brown")

)

# Sample 4 elements from the 'fruits' list with replacement sampled\_list<- sample(population\_list$fruits, size = 4, replace = TRUE) print(sampled\_list)

#### Output

"Apple" "Cherry" "Apple" "Date"

#### Replicating Sampling:

# Define a population vector

population\_vector<- c(10, 20, 30, 40, 50, 60, 70, 80, 90, 100)

# Replicate sampling 5 times without replacement

replicated\_samples<- replicate(5, sample(population\_vector, size = 3, replace = FALSE))

# Print the replicated samples print(replicated\_samples)

Output

[,1] [,2] [,3] [,4] [,5]

[1,] 60 80 50 90 10

[2,] 80 20 100 70 100

[3,] 10 10 60 60 20

#### Sampling without replacement

In this case of without replacement, the function **replace=F** is used and it will not allow the repetition of values.

#samples without replacement x<-sample(1:8,7, replace=F) x

**Output** ->4165327

x<-sample(1:8,9, replace=F)

Error insample.int(length(x), size, replace,prob):

cannot take a sample larger than the population when 'replace = FALSE' #here the size of the sample is equal to range 'x'.

x<-sample(1:5,5, replace=F) x

**Output** ->54132

#### Sampling from a vector without replacement

# Sampling from a vector without replacement items<- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

sample\_size<- 5

sample<- sample(items, size = sample\_size, replace = FALSE) print(sample)

Output 3 5 6 9 8

1. Shuffling a deck of cards (52 cards) without replacement deck<- 1:52

shuffled\_deck<- sample(deck, size = length(deck), replace = FALSE)

hand\_size<- 5

hand<- shuffled\_deck[1:hand\_size] print(hand)

Output

9 34 41 43 11

#### Random sampling using the dplyr package

The dplyr package is a well known R package for data manipulation and transformation. It gives a bunch of functions that make it simpler to work with data casings and data tables in R. One common undertaking in data analysis is random sampling, which can be accomplished using the sample\_n() and sample\_frac() functions in dplyr.

1: Randomly Sampling Rows from a Data Frame # Load the dplyr package

library(dplyr)

# Create a sample data frame # Set seed for reproducibility set.seed(123)

data<- data.frame( ID = 1:100,

Value = rnorm(100)

)

# Randomly sample 10 rows from the data frame sampled\_data<- data %>%

sample\_n(10)

# View the sampled data print(sampled\_data)

Output

ID Value

1 7 0.56047565

2 9 -0.23017749

3 15 1.55870831

4 16 0.07050839

5 20 1.71506598

6 23 -0.68685285

7 42 1.78691314

8 46 1.06782371

9 50 0.49850701

10 68 -0.29472045

2: Random Sampling a Fraction of Rows from a Data Frame # Load the dplyr package

library(dplyr)

# Create a sample data frame # Set seed for reproducibility set.seed(456)

data<- data.frame( ID = 1:200,

Value = rnorm(200)

)

# Randomly sample 20% of the rows from the data frame sampled\_data<- data %>%

sample\_frac(0.20)

# View the sampled data print(sampled\_data)

Output

ID Value

1 151 1.200410172

2 140 -0.181812198

3 88 0.920529800

4 68 -1.431378346

5 191 -0.697237001

6 27 -0.462854969

7 75 -0.020014663

8 90 -0.236867797

9 46 0.120851803

10 71 -0.169987994

11 163 -1.035274763

12 62 -0.982060062

13 175 -1.549384356

14 85 0.708817307

15 174 0.309910662

16 119 -1.433778349

17 49 -1.175402402

18 126 -1.126327533

19 69 -0.544594202

20 130 0.355610384

21 193 1.232308978

22 36 1.815652319

23 60 0.577150467

24 132 1.149194486

25 118 1.207347447

26 42 0.393037377

27 131 0.004052138

28 167 1.772544877

29 181 -1.388188492

30 45 2.078874614

31 17 1.736936177

32 77 -0.112933852

33 26 1.134284565

34 124 0.313843454

35 133 -0.496614335

36 83 2.020634788

37 35 0.170625252

38 197 -0.196112610

39 116 0.982940735

40 149 1.210757937

**[Taking samples using the function set.seed()](https://www.digitalocean.com/community/tutorials/sample-in-r" \l "taking-samples-using-the-function-set-seed)**

**set.seed()** - set.seed function will produce the same sequence when you run it. #set the index

set.seed(5)

#takes the random samples with replacement sample(1:5,4, replace=T)

2313

set.seed(5)

sample(1:5,4, replace=T) 2313

set.seed(5)

sample(1:5,4, replace=T) 2313

[Taking the samples from the dataset using the set.seed() function](https://www.digitalocean.com/community/tutorials/sample-in-r" \l "taking-the-samples-from-the-dataset-using-the-set-seed-function)

#set.seed function set.seed(10)

#taking sample of 10 rows from the iris dataset. x<- sample(1:nrow(iris), 10)

x

-->137741127288151431492413

#displays the 10 rows iris[x, ]

#### Output:

Sepal.LengthSepal.WidthPetal.LengthPetal.Width Species 1376.33.45.62.4virginica

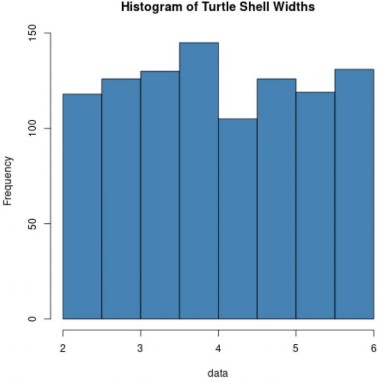
746.12.84.71.2versicolor 1126.42.75.31.9virginica 726.12.84.01.3versicolor 886.32.34.41.3versicolor 155.84.01.20.2setosa 1435.82.75.11.9virginica 1496.23.45.42.3virginica 245.13.31.70.5setosa 134.83.01.40.1setosa

#### CENTRAL LIMIT THEOREM IN R

#make this example reproducible set.seed(0)

#create random variable with sample size of 1000 that is uniformally distributed data<- runif(n=1000, min=2, max=6)

#create histogram to visualize distribution of turtle shell widths hist(data, col='steelblue', main='Histogram of Turtle Shell Widths')



#create empty vector to hold sample means sample5 <- c()

#take 1,000 random samples of size n=5 n = 1000

for(i in 1:n){

sample5[i] = mean(sample(data, 5, replace=TRUE))

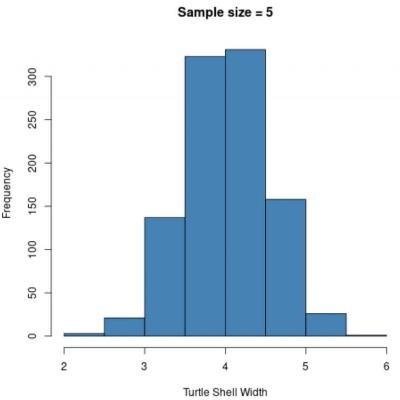
}

#calculate mean and standard deviation of sample means

mean(sample5) [1] 4.008103

sd(sample5) [1] 0.5171083

#create histogram to visualize sampling distribution of sample means hist(sample5, col ='steelblue', xlab='Turtle Shell Width', main='Sample size = 5')



Notice that the sampling distribution of sample means appears normally distributed, even though the distribution that the samples came from was not normally distributed. Also notice the sample mean and sample standard deviation for this sampling distribution:

Now suppose the sample size is increased from n=5 to n=30 and recreate the histogram of sample means:

**#**create empty vector to hold sample means sample30 <- c()

#take 1,000 random samples of size n=30 n = 1000

for(i in 1:n){

sample30[i] = mean(sample(data, 30, replace=TRUE))

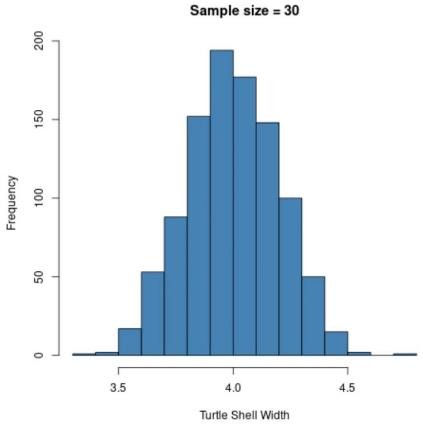
}

#calculate mean and standard deviation of sample means mean(sample30)

[1] 4.000472

sd(sample30) [1] 0.2003791

#create histogram to visualize sampling distribution of sample means hist(sample30, col ='steelblue', xlab='Turtle Shell Width', main='Sample size = 30')



The sampling distribution is normally distributed once again, but the sample standard deviation is even smaller:

s: 0.200

This is because, a larger sample size (n = 30) is used compared to the previous example (n = 5) so the standard deviation of sample means is even smaller.

If using larger and larger sample sizes are used , the sample standard deviation gets smaller and smaller. This illustrates the central limit theorem in practice.

Applying the Central Limit Theorem in R

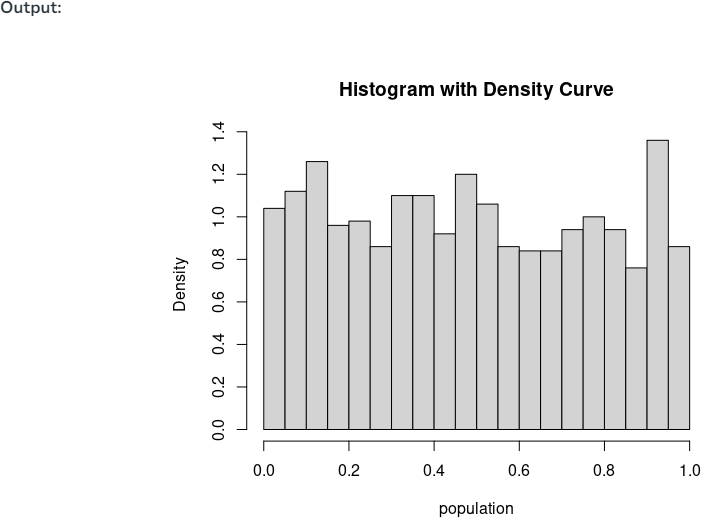
1. Generate a Non-Normally Distributed Population

# Generate a non-normally distributed population set.seed(42)

population<- runif(1000, min = 0, max = 1)

# Create a histogram of the population

hist(population, breaks = 20, probability = TRUE, main = "Histogram with Density Curve")



1. Draw Random Samples

To draw multiple random samples from this population.

The sample size should be large enough for the CLT to hold (typically, a sample size of at least 30 is considered ).

# Set the sample size and number of samples sample\_size<- 20

num\_samples<- 500

# Draw random samples

samples<- replicate(num\_samples, sample(population, size = sample\_size,replace = TRUE))

1. Check mean and Variance of Sample Mean and Populations # Calculate sample means

sample\_means<- colMeans(samples)

# For sample

x\_bar<- mean(sample\_means) std<- sd(sample\_means)

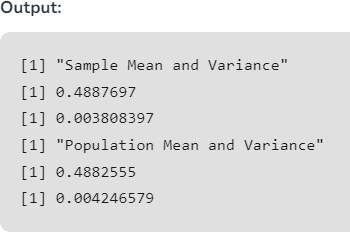
print('Sample Mean and Variance') print(x\_bar)

print(std\*\*2)

# For Population

mu<- mean(population) sigma<- sd(population)

print('Population Mean and Variance') print(mu) print((sigma\*\*2)/sample\_size)



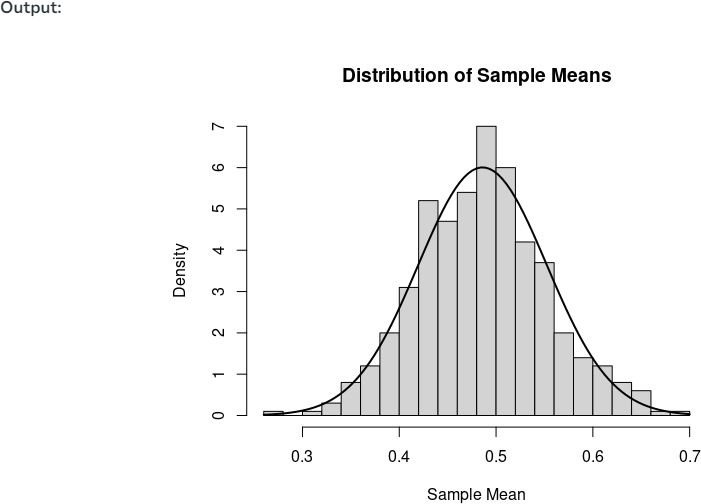
1. Plot the Sample distributions

# Visualize the sample means

hist(sample\_means, breaks = 15, prob = TRUE, main = "Distribution of Sample Means", xlab = "Sample Mean")

# Distribution Curve

curve(dnorm(x, mean = x\_bar, sd = std), col = "Black", lwd = 2, add = TRUE)



The resulting plot show that the distribution of sample means closely follows a normal distribution, even though the original population was not normally distributed. This is a direct demonstration of the Central Limit Theorem in action.

Example 2: Central limit theorem in R

# Set the random seed for reproducibility set.seed(42)

# Generate a non-normally distributed population population<- runif(5000, min = 0, max = 1)

# Create a histogram of the population

par(mfrow = c(1, 2)) # Set up a 1x2 grid for plotting

# Plot the histogram of the population

hist(population, breaks = 30, prob = TRUE, main = "Population Distribution", xlab = "Value", col = "lightblue")

# Step 2 and 3: Draw random samples and calculate sample means sample\_size<- 30

num\_samples<- 300

# Empty vector to store sample means sample\_means<- c()

for (i in 1:num\_samples) { # Take a random sample

sample<- sample(population, size = sample\_size, replace = TRUE)

# Calculate the mean of the sample sample\_means[i] <- mean(sample)

}

# For sample

x\_bar<- mean(sample\_means) std<- sd(sample\_means)

print('Sample Mean and Variance') print(x\_bar)

print(std\*\*2)

# For Population

mu<- mean(population) sigma<- sd(population)

print('Population Mean and Variance') print(mu) print((sigma\*\*2)/sample\_size)

# Plot the histogram of sample means

hist(sample\_means, breaks = 30, prob = TRUE, main = "Distribution of Sample Means", xlab = "Sample Mean", col = "lightgreen")

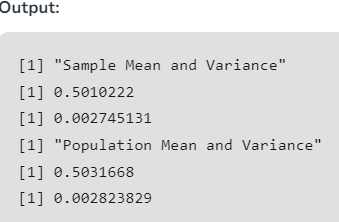
# Overlay density curves

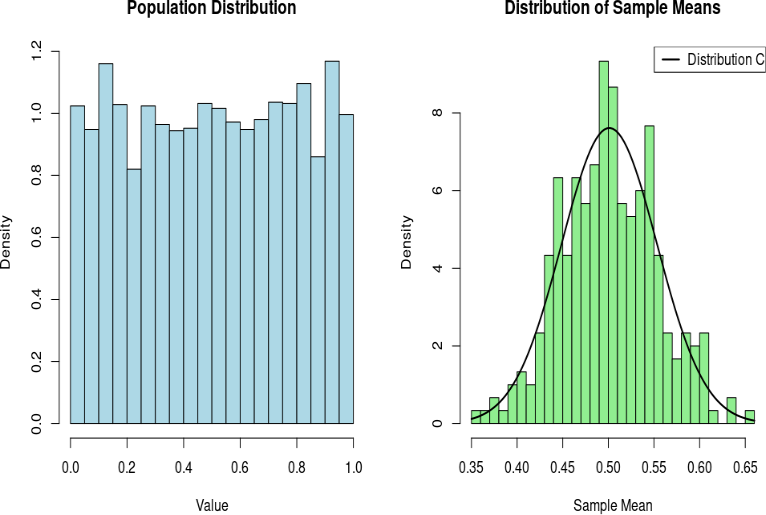
curve(dnorm(x, mean = x\_bar, sd = std), col = "black", lwd = 2, add = TRUE)

# Add labels and legends

legend("topright", legend = c("Distribution Curve"), col = c("black"), lwd = 2)

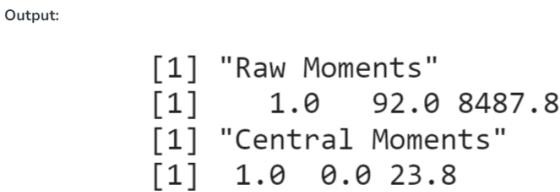
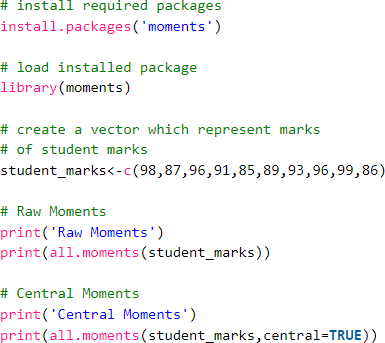
# Reset the plot layout par(mfrow = c(1, 1))





## COMPUTE RAW AND CENTRAL MOMENTS USING R

#### Program



**Skewness and Kurtosis in R Programming**

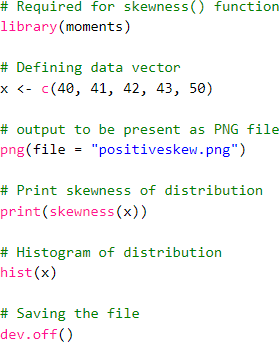
Skewness

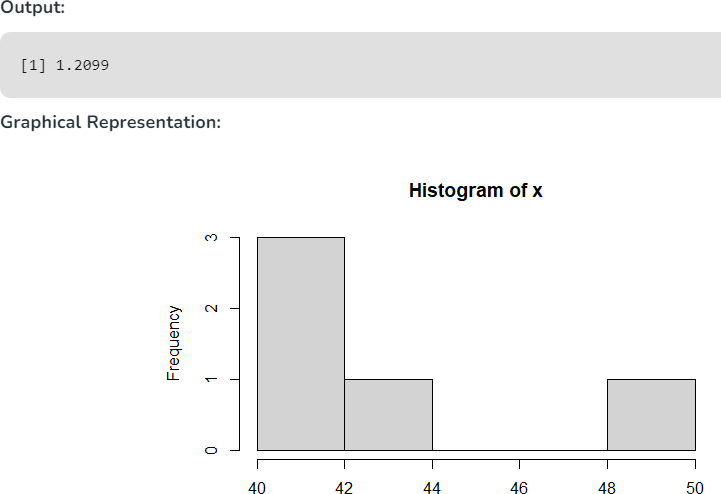
Skewness is a statistical numerical method to measure the asymmetry of the distribution or data set. It tells about the position of the majority of data values in the distribution around the mean value. A fundamental statistical notion called skewness quantifies the asymmetries in data distributions

There exist 3 types of skewness values on the basis of which the asymmetry of the graph is decided. These are as follows:

#### Positive Skew

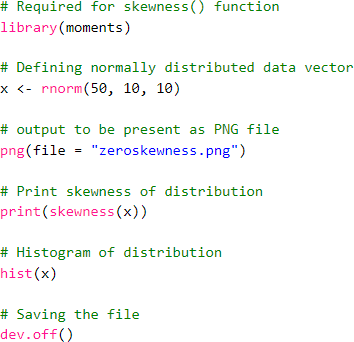
The asymmetry of data distributions where the tail extends towards higher values is known statistically as positive skewness. If the coefficient of skewness is greater than 0, then the graph is said to be positively skewed with the majority of data values less than the mean.

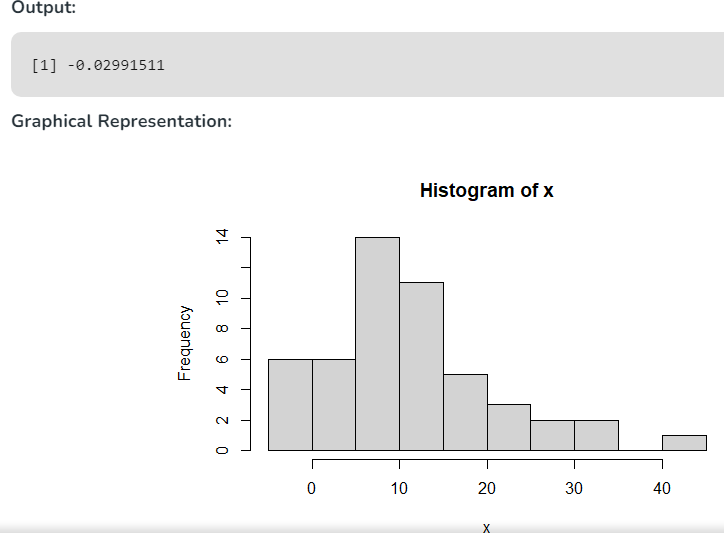




#### Zero Skewness or Symmetric

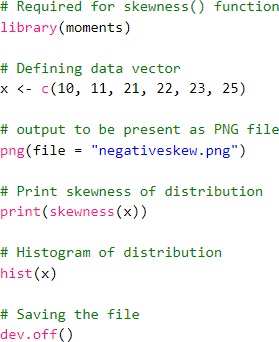
A statistical notion called zero skewness, commonly referred to as symmetry, defines data distributions that are balanced and have equal probability on both sides of the mean. If the coefficient of skewness is equal to 0 or approximately close to 0, then the graph is said to be symmetric and data is normally distributed.



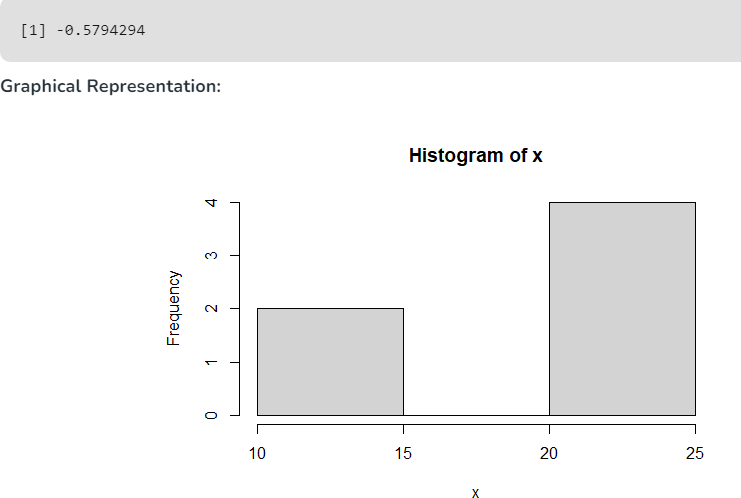


#### Negatively skewed

Left-skewed distributions, commonly referred to as negatively skewed distributions, are statistical notions that describe asymmetrical data distributions with a tail that slopes downward. If the coefficient of skewness is less than 0, then the graph is said to be negatively skewed with the majority of data values greater than the mean. Most of the values are concentrated on the right side of the graph.



Output:



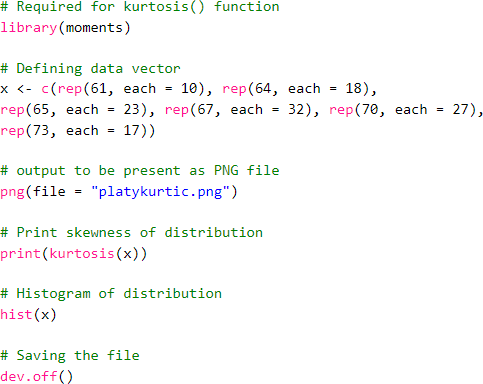
Kurtosis

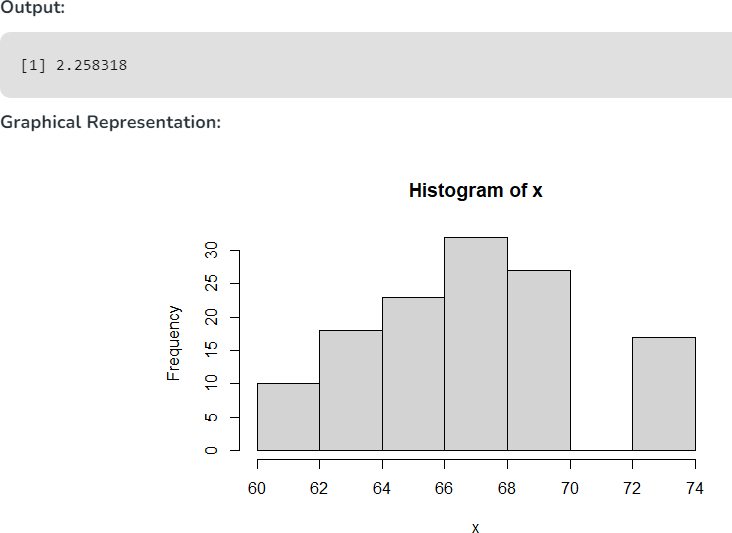
A statistical measure known as kurtosis measures the peakedness, flatness, and weight of the tails of data distributions. Kurtosis is a numerical method in statistics that measures the sharpness of the peak in the data distribution.

There exist 3 types of Kurtosis values on the basis of which the sharpness of the peak is measured. These are as follows:

#### Platykurtic

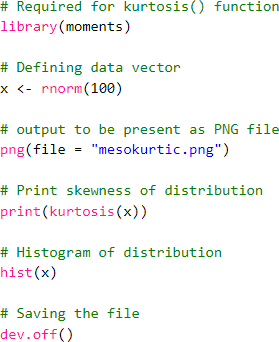
Data distributions having flattened tails compared to the normal distribution are referred to statistically as platykurtic distributions. If the coefficient of kurtosis is less than 3, then the data distribution is platykurtic. Being platykurtic doesn’t mean that the graph is flat-topped.

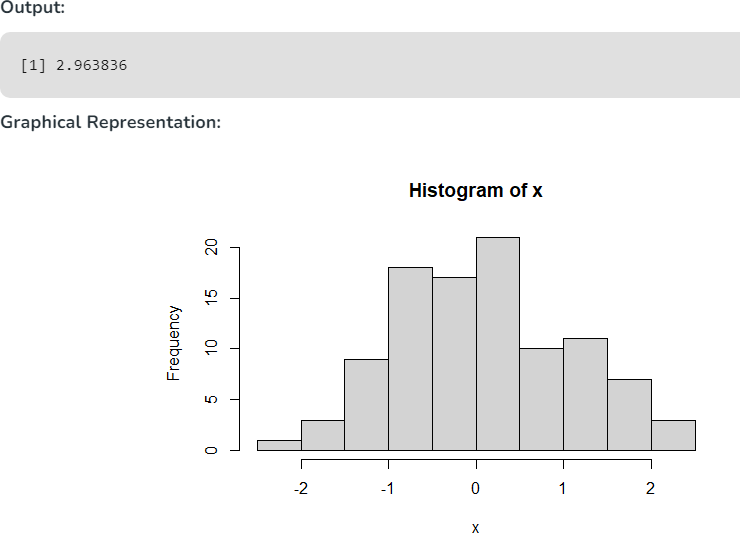




Mesokurtic

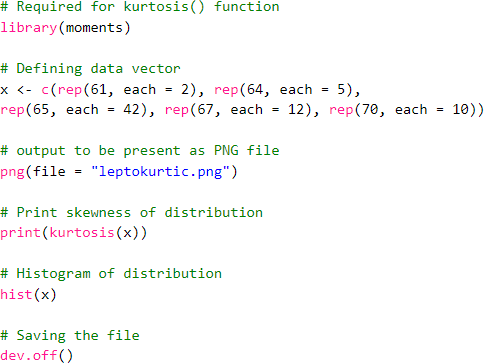
Data distributions with tails that are similar in thickness to the normal distribution are known statistically as mesokurtic distributions. If the coefficient of kurtosis is equal to 3 or approximately close to 3, then the data distribution is mesokurtic. For the normal distribution, the kurtosis value is approximately equal to 3.

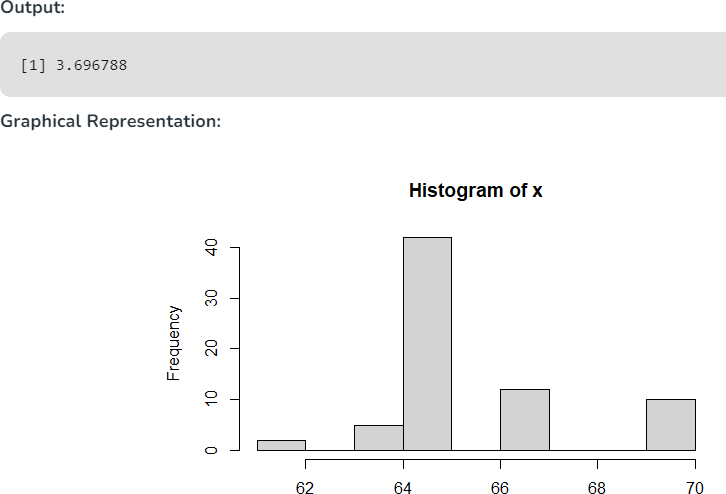




Leptokurtic

Data distributions having hefty tails compared to the normal distribution are referred to statistically as leptokurtic distributions. If the coefficient of kurtosis is greater than 3, then the data distribution is leptokurtic and shows a sharp peak on the graph.



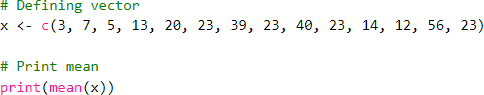


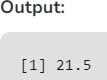
## MEASURES OF THE CENTRAL TENDENCY

#### Program:

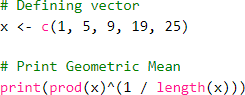
Central tendency tells about how the group of data is clustered around the center value of the distribution. Central tendency performs the following measures:

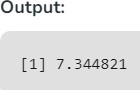
#### Arithmetic Mean



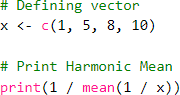


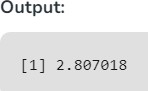
**Geometric Mean**



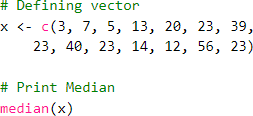


#### Harmonic Mean



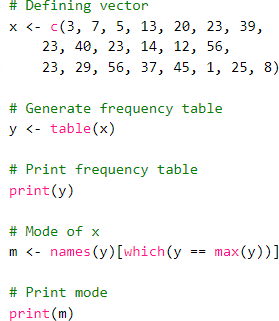


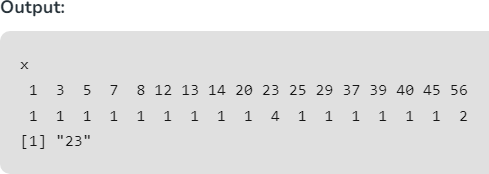
**Median**



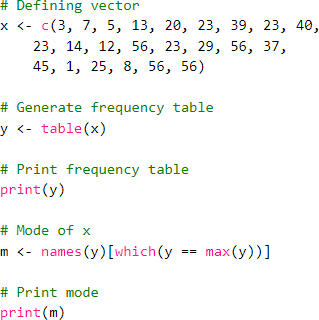


Mode





**Example 2: Multiple Mode values**



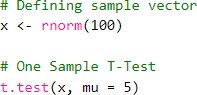


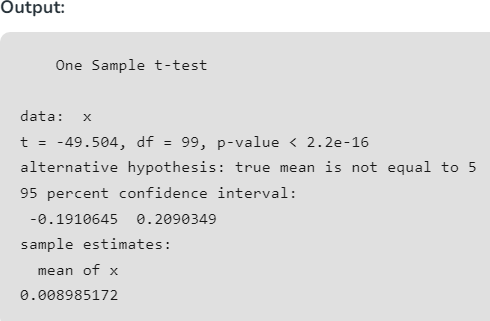
## STATISTICAL TESTS OF SIGNIFICANCE

#### T-Test Approach in R Programming

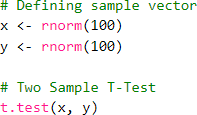
**One Sample T – Test Approach**

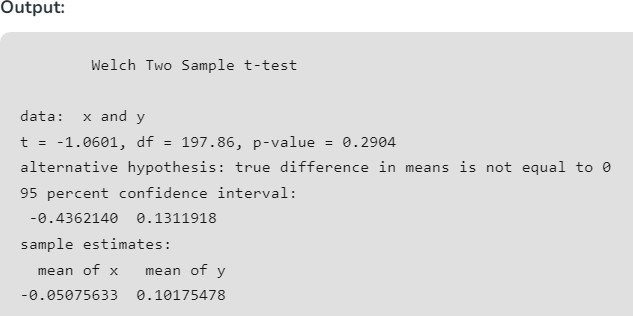
The One-Sample T-Test is used to test the statistical difference between a sample mean and a known or assumed/hypothesized value of the mean in the population.





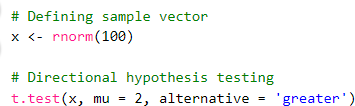
Two Sample T-Testing

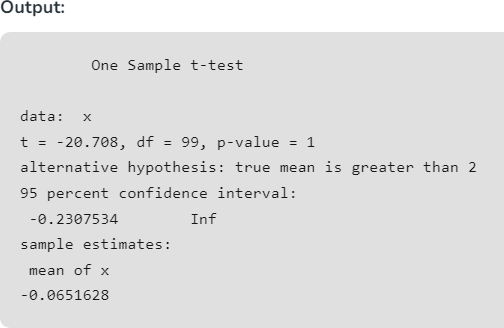




#### Hypothesis Testing in R Programming

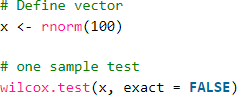
A hypothesis is an assumption made by the researchers that are not mandatory true. To perform hypothesis testing, a random sample of data from the population is taken and testing is performed. Based on the results of the testing, the hypothesis is either selected or rejected. This concept is known as Statistical Inference.

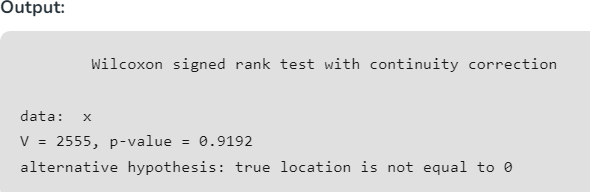




***One Sample -Test***

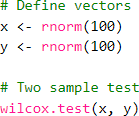
# Sample Test

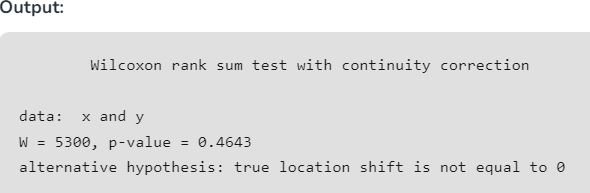




##### Two Sample -Test

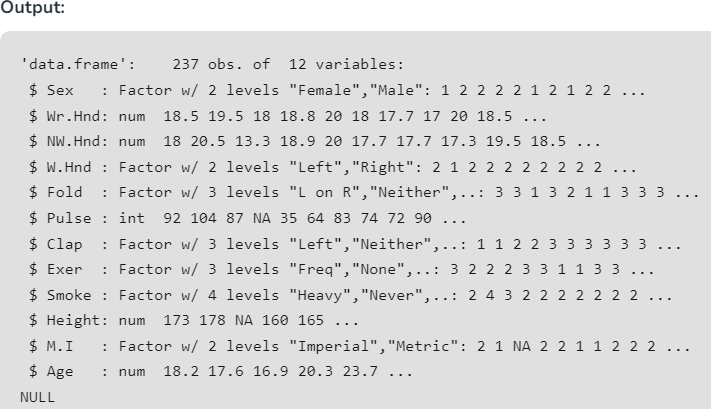
This test is performed to compare two samples of data



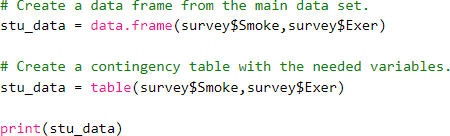


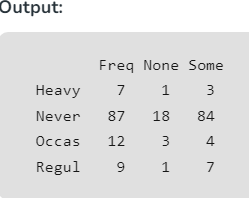
# Chi-Square Test in R



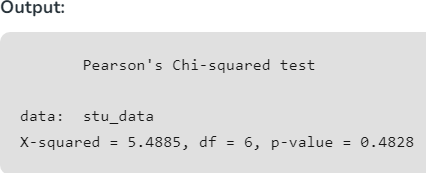


The above result shows the dataset has many factor variables which can be considered as categorical variables. Consider the variables “Exer” and “Smoke“. The Smoke column records the students smoking habits while the Exer column records their exercise level. The aim is to test the hypothesis whether the students smoking habit is independent of their exercise level at .05 significance level.

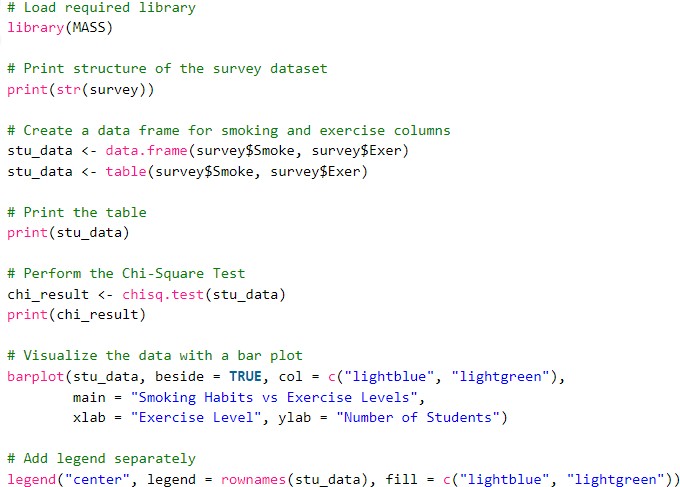


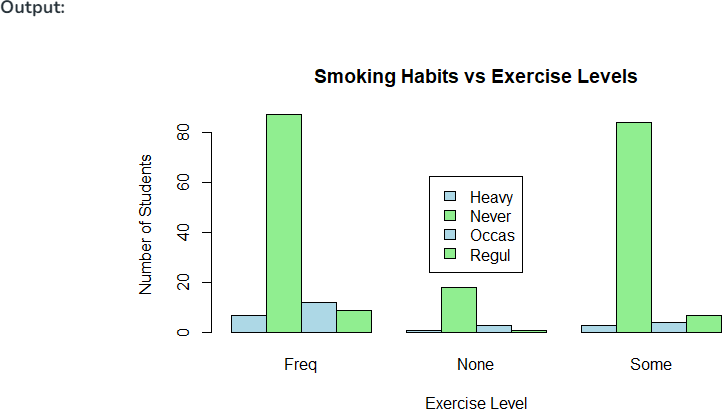


And finally, apply the chisq.test() function to the contingency table stu\_data.



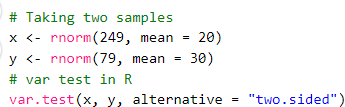
**Visualize the Chi-Square Test data**

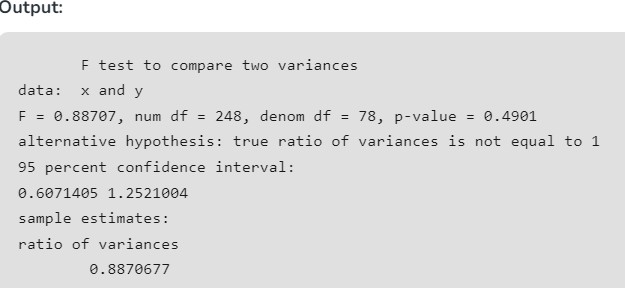




# Fisher’s F-Test in R Programming

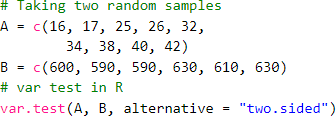
Fisher’s F test calculates the ratio between the larger variance and the smaller variance.



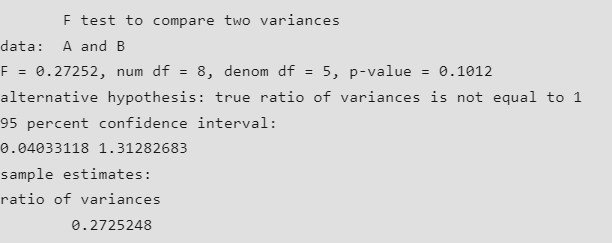


Example 2:

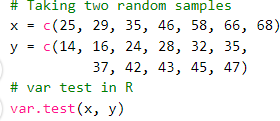
Let us have two random samples from two random populations. Test whether two populations have the same variance.

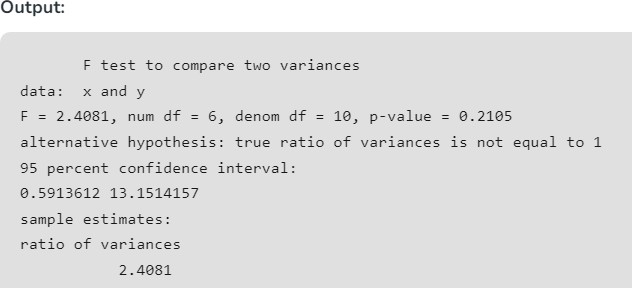


OUTPUT:



Consider two random samples





1. LINEAR REGRESSION

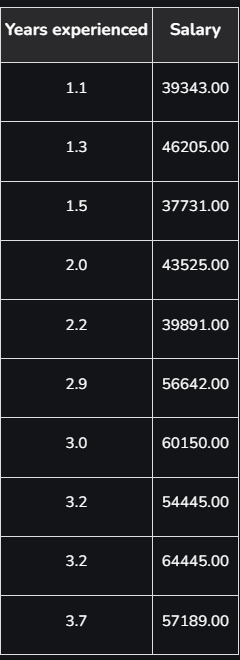
Program

Regression shows a line or curve that passes through all the data points on the target-predictor graph in such a way that the vertical distance between the data points and the regression line is minimum

Linear Regression is a statistical approach for modelling the relationship between a dependent variable and a given set of independent variables. It is predicted that a straight line can be used to approximate the relationship. The goal of linear regression is to identify the line that minimizes the discrepancies between the observed data points and the line’s anticipated values.

For example, consider a salary dataset where it is given the value of the dependent variable(salary) for every independent variable(years experienced).

Salary dataset:



For general purposes, define:

* x as a feature vector, i.e x = [x\_1, x\_2, …., x\_n],
* y as a response vector, i.e y = [y\_1, y\_2, …., y\_n]
* for n observations (in the above example, n=10).

1. First, convert these data values into R Data Frame

# Create the data frame data <- data.frame(

Years\_Exp = c(1.1, 1.3, 1.5, 2.0, 2.2, 2.9, 3.0, 3.2, 3.2, 3.7),

Salary = c(39343.00, 46205.00, 37731.00, 43525.00,

39891.00, 56642.00, 60150.00, 54445.00, 64445.00, 57189.00)

)

1. Scatter plot of the given dataset

Output:

# Create the scatter plot plot(data$Years\_Exp, data$Salary,

xlab = "Years Experienced", ylab = "Salary",

main = "Scatter Plot of Years Experienced vs Salary")



Now, find a line that fits the above scatter plot through which can predict any value of y or response for any value of x. The line which best fits is called the Regression line. The equation of the regression line is given by:

y = a + bx

Where y is the predicted response value, a is the y-intercept, x is the feature value and b is the slope.

1. R code is used to implement Linear Regression

install.packages('caTools') library(caTools)

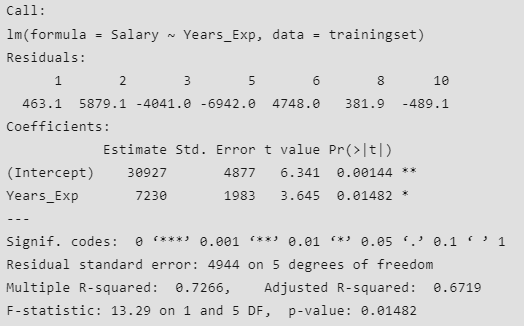
split = sample.split(data$Salary, SplitRatio = 0.7) trainingset = subset(data, split == TRUE)

testset = subset(data, split == FALSE)

# Fitting Simple Linear Regression to the Training set lm.r= lm(formula = Salary ~ Years\_Exp,

data = trainingset) #Summary of the model summary(lm.r)

Output:



1. Predict values using predict function # Create a data frame with new input values

new\_data<- data.frame(Years\_Exp = c(4.0, 4.5, 5.0))

# Predict using the linear regression model predicted\_salaries<- predict(lm.r, newdata = new\_data)

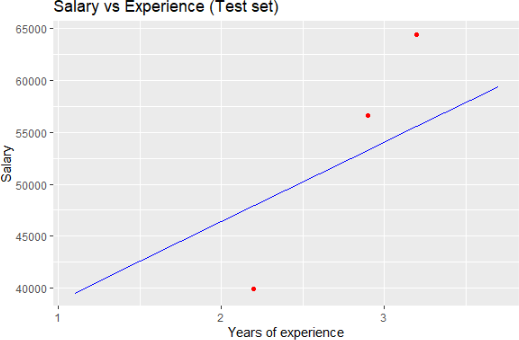
# Display the predicted salaries print(predicted\_salaries)

Output:



|  |
| --- |
| 5. Visualizing the Training set results:  # Visualising the Training set results  ggplot() + geom\_point(aes(x = trainingset$Years\_Ex, y = trainingset$Salary), colour = 'red') + geom\_line(aes(x = trainingset$Years\_Ex, y = predict(lm.r, newdata = trainingset)), colour = 'blue') + ggtitle('Salary vs Experience (Training set)') +  xlab('Years of experience') + ylab('Salary')  Output: |
| # Visualising the Test set results ggplot() +  geom\_point(aes(x = testset$Years\_Exp, y = testset$Salary),  colour = 'red') + geom\_line(aes(x = trainingset$Years\_Exp,  y = predict(lm.r, newdata = trainingset)), colour = 'blue') +  ggtitle('Salary vs Experience (Test set)') + xlab('Years of experience') + ylab('Salary') |

Output:



#### 9. REGULARIZATIONS

**Program:**

Regularization is a form of regression technique that shrinks or regularizes or constraints the coefficient estimates towards 0 (or zero). In this technique, a penalty is added to the various parameters of the model in order to reduce the freedom of the given model.

In R language, to perform Regularization, a handful of packages are needed to be installed before start working on them. The required packages are

* glmnet package for ridge regression and lasso regression
* dplyr package for data cleaning
* psych package in order to perform or compute the trace function of a matrix
* caret package

To install these packages, use the install.packages() in the R Console. After installing the packages successfully, include these packages in R Script using the library() command. To implement the Regularization regression technique either of the three types of regularization techniques are used.

#### Ridge Regression

The Ridge Regression is a modified version of linear regression and is also known as L2 Regularization. Unlike linear regression, the loss function is modified in order to minimize the model’s complexity and this is done by adding some penalty parameter which is equivalent to the square of the value or magnitude of the coefficient. Basically, to implement Ridge Regression in R “glmnet” package is used. The cv.glmnet() function will be used to determine the ridge regression.

Example:

To implement the ridge regression technique on the mtcars dataset. The task is to predict the miles per gallon on the basis of other characteristics of the cars. Use the set.seed() function to set seed for reproducibility.

# Regularization

# Ridge Regression in R

# Load libraries, get data & set # seed for reproducibility set.seed(123)

library(glmnet) library(dplyr) library(psych)

data("mtcars")

# Center y, X will be standardized # in the modelling function

y <- mtcars %>% select(mpg) %>%

scale(center = TRUE, scale = FALSE) %>% as.matrix()

X <- mtcars %>% select(-mpg) %>% as.matrix()

# Perform 10-fold cross-validation to select lambda lambdas\_to\_try<- 10^seq(-3, 5, length.out = 100)

# Setting alpha = 0 implements ridge regression ridge\_cv<- cv.glmnet(X, y, alpha = 0,

lambda = lambdas\_to\_try, standardize = TRUE, nfolds = 10)

# Plot cross-validation results plot(ridge\_cv)

# Best cross-validated lambda lambda\_cv<- ridge\_cv$lambda.min

# Fit final model, get its sum of squared # residuals and multiple R-squared

model\_cv<- glmnet(X, y, alpha = 0, lambda = lambda\_cv,

standardize = TRUE) y\_hat\_cv<- predict(model\_cv, X)

ssr\_cv<- t(y - y\_hat\_cv) %\*% (y - y\_hat\_cv) rsq\_ridge\_cv<- cor(y, y\_hat\_cv)^2

# selecting lambda based on the information X\_scaled<- scale(X)

aic<- c()

bic<- c()

for (lambda in seq(lambdas\_to\_try))

{

# Run model

model<- glmnet(X, y, alpha = 0,

lambda = lambdas\_to\_try[lambda], standardize = TRUE)

# Extract coefficients and residuals (remove first # row for the intercept)

betas<- as.vector((as.matrix(coef(model))[-1, ])) resid<- y - (X\_scaled %\*% betas)

# Compute hat-matrix and degrees of freedom

ld<- lambdas\_to\_try[lambda] \* diag(ncol(X\_scaled))

H <- X\_scaled %\*% solve(t(X\_scaled) %\*% X\_scaled + ld) %\*% t(X\_scaled) df<- tr(H)

# Compute information criteria

aic[lambda] <- nrow(X\_scaled) \* log(t(resid) %\*% resid) + 2 \* df bic[lambda] <- nrow(X\_scaled) \* log(t(resid) %\*% resid)

+ 2 \* df \* log(nrow(X\_scaled))

}

# Plot information criteria against tried values of lambdas plot(log(lambdas\_to\_try), aic, col = "orange", type = "l",

ylim = c(190, 260), ylab = "Information Criterion") lines(log(lambdas\_to\_try), bic, col = "skyblue3")

legend("bottomright", lwd = 1, col = c("orange", "skyblue3"), legend = c("AIC", "BIC"))

# Optimal lambdas according to both criteria lambda\_aic<- lambdas\_to\_try[which.min(aic)] lambda\_bic<- lambdas\_to\_try[which.min(bic)]

# Fit final models, get their sum of

# squared residuals and multiple R-squared

model\_aic<- glmnet(X, y, alpha = 0, lambda = lambda\_aic,

standardize = TRUE)

y\_hat\_aic<- predict(model\_aic, X)

ssr\_aic<- t(y - y\_hat\_aic) %\*% (y - y\_hat\_aic) rsq\_ridge\_aic<- cor(y, y\_hat\_aic)^2

model\_bic<- glmnet(X, y, alpha = 0, lambda = lambda\_bic,

standardize = TRUE)

y\_hat\_bic<- predict(model\_bic, X)

ssr\_bic<- t(y - y\_hat\_bic) %\*% (y - y\_hat\_bic) rsq\_ridge\_bic<- cor(y, y\_hat\_bic)^2

# The higher the lambda, the more the

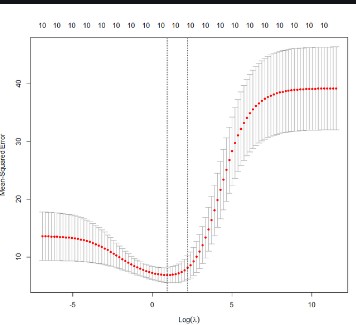
# coefficients are shrinked towards zero.

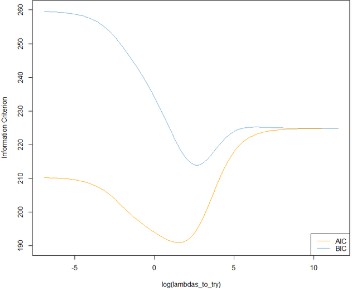
res<- glmnet(X, y, alpha = 0, lambda = lambdas\_to\_try,

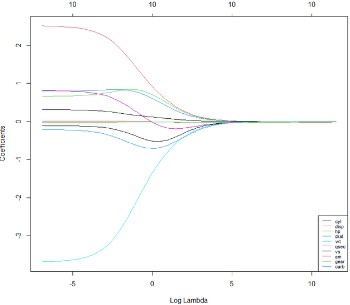
standardize = FALSE) plot(res, xvar = "lambda") legend("bottomright", lwd = 1, col = 1:6,

legend = colnames(X), cex = .7)

#### Output:







1. **Lasso Regression**

It is also known as L1 Regression, Selection Operator, and Least Absolute Shrinkage. It is also a modified version of Linear Regression where again the loss function is modified in order to minimize the model’s complexity. This is done by limiting the summation of the absolute values of the coefficients of the model. In R, the lasso regression can be implemented using the same “glmnet” package like ridge regression.

Program:

# Regularization

# Lasso Regression

# Load libraries, get data & set # seed for reproducibility set.seed(123)

library(glmnet) library(dplyr) library(psych)

data("mtcars")

# Center y, X will be standardized in the modelling function

y <- mtcars %>% select(mpg) %>% scale(center = TRUE, scale = FALSE) %>% as.matrix() X <- mtcars %>% select(-mpg) %>% as.matrix()

# Perform 10-fold cross-validation to select lambda lambdas\_to\_try<- 10^seq(-3, 5, length.out = 100)

# Setting alpha = 1 implements lasso regression lasso\_cv<- cv.glmnet(X, y, alpha = 1,

lambda = lambdas\_to\_try, standardize = TRUE, nfolds = 10)

# Plot cross-validation results plot(lasso\_cv)

# Best cross-validated lambda lambda\_cv<- lasso\_cv$lambda.min

# Fit final model, get its sum of squared # residuals and multiple R-squared

model\_cv<- glmnet(X, y, alpha = 1, lambda = lambda\_cv,

standardize = TRUE) y\_hat\_cv<- predict(model\_cv, X)

ssr\_cv<- t(y - y\_hat\_cv) %\*% (y - y\_hat\_cv) rsq\_lasso\_cv<- cor(y, y\_hat\_cv)^2

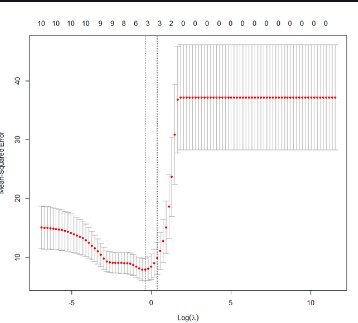
# The higher the lambda, the more the

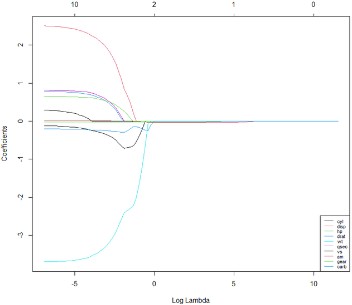
# coefficients are shrinked towards zero.

res<- glmnet(X, y, alpha = 1, lambda = lambdas\_to\_try,

standardize = FALSE) plot(res, xvar = "lambda") legend("bottomright", lwd = 1, col = 1:6,

legend = colnames(X), cex = .7)





#### Elastic Net Regression

Elastic Net Regression can be stated as the convex combination of the lasso and ridge regression. Program:

# Regularization

# Elastic Net Regression library(caret)

# Set training control

train\_control<- trainControl(method = "repeatedcv",

number = 5,

repeats = 5,

search = "random", verboseIter = TRUE)

# Train the model elastic\_net\_model<- train(mpg ~ .,

data = cbind(y, X), method = "glmnet",

preProcess = c("center", "scale"), tuneLength = 25,

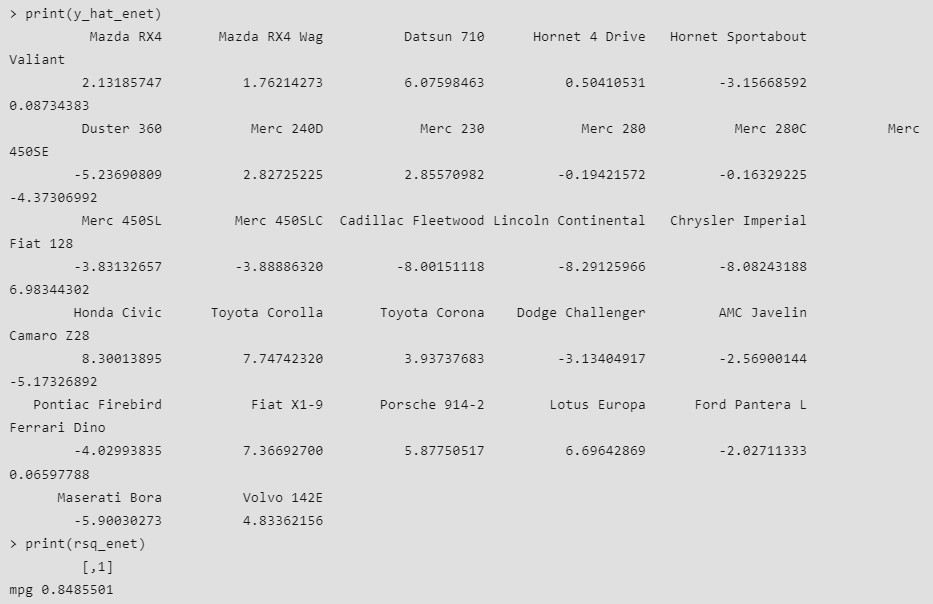
trControl = train\_control)

# Check multiple R-squared

y\_hat\_enet<- predict(elastic\_net\_model, X) rsq\_enet<- cor(y, y\_hat\_enet)^2 print(y\_hat\_enet)

print(rsq\_enet)

#### Output:



**10. HYPOTHESIS TESTING**

#### Program:

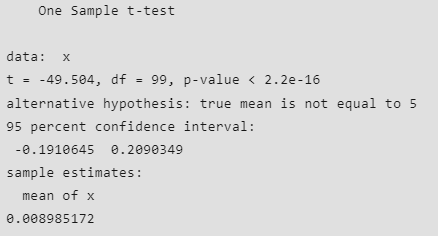
To perform hypothesis testing, a random sample of data from the population is taken and testing is performed. Based on the results of the testing, the hypothesis is either selected or rejected. This concept is known as Statistical Inference.

1. One sample T-Testing

One sample T-Testing approach collects a huge amount of data and tests it on random samples. To perform T-Test in R, normally distributed data is required. This test is used to test the mean of the sample with the population. For example, the height of persons living in an area is different or identical to other persons living in other areas.

# Defining sample vector x <- rnorm(100)

# One Sample T-Test t.test(x, mu = 5) **Output:**



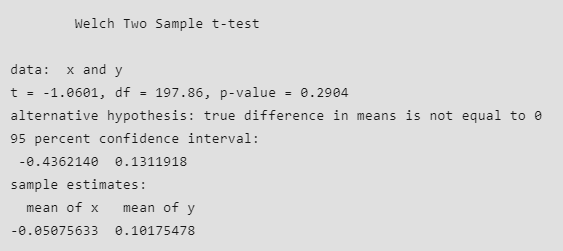
#### Two-sample T-Testing

In two sample T-Testing, the sample vectors are compared. If var. equal = TRUE, the test assumes that the variances of both the samples are equal.

# Defining sample vector x <- rnorm(100)

y <- rnorm(100)

# Two Sample T-Test t.test(x, y)

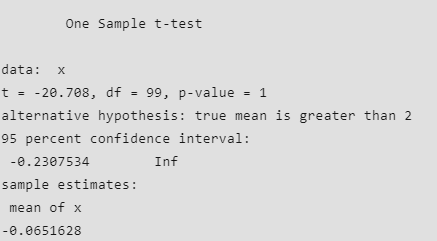


#### Directional Hypothesis

Using the directional hypothesis, the direction of the hypothesis can be specified like, if the user wants to know the sample mean is lower or greater than another mean sample of the data.

# Defining sample vector x <- rnorm(100)

# Directional hypothesis testing t.test(x, mu = 2, alternative = 'greater')

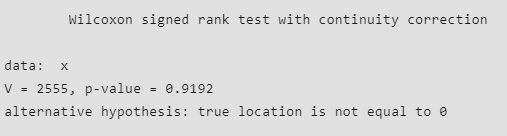


#### One sample µ-test

This type of test is used when comparison has to be computed on one sample and the data is non- parametric. It is performed using wilcox.test() function in R programming.

# Define vector x <- rnorm(100)

# one sample test wilcox.test(x, exact = FALSE)

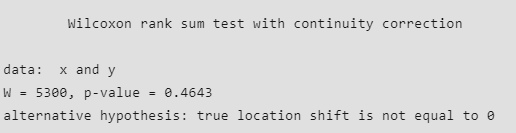


#### Two sample µ-test

This test is performed to compare two samples of data. # Define vectors

x <- rnorm(100) y <- rnorm(100)

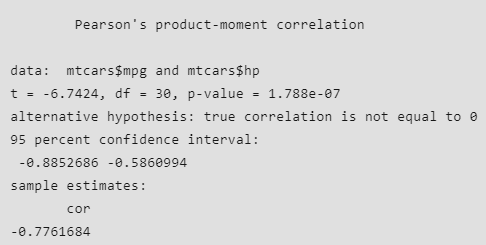
# Two sample test wilcox.test(x, y)



#### Correlation test

This test is used to compare the correlation of the two vectors provided in the function call or to test for the association between the paired samples.

# Usingmtcars dataset in R cor.test(mtcars$mpg, mtcars$hp)



#### STATISTICAL ANALYSIS

**Program**:

# Load the iris data set data(iris)

# Select the variables to include in the PCA analysis vars <- c("Sepal.Length", "Sepal.Width",

"Petal.Length", "Petal.Width")

# Subset the data to include only the selected variables data\_subset <- iris[, vars]

# Scale the data

data\_scaled <- scale(data\_subset) # Perform PCA

pca <- prcomp(data\_scaled,

center = TRUE, scale. = TRUE) # Print the summary of the PCA results

summary(pca)

**Output:**

Importance of components:

PC1 PC2 PC3 PC4

### Standard deviation 1.7084 0.9560 0.38309 0.14393

Proportion of Variance 0.7296 0.2285 0.03669 0.00518

### Cumulative Proportion 0.7296 0.9581 0.99482 1.00000

**Different Visualizations for the dataset** # Load the ggplot2 library library(ggplot2)

# Generate some sample data data <- data.frame(

var1 = rnorm(100), var2 = rnorm(100),

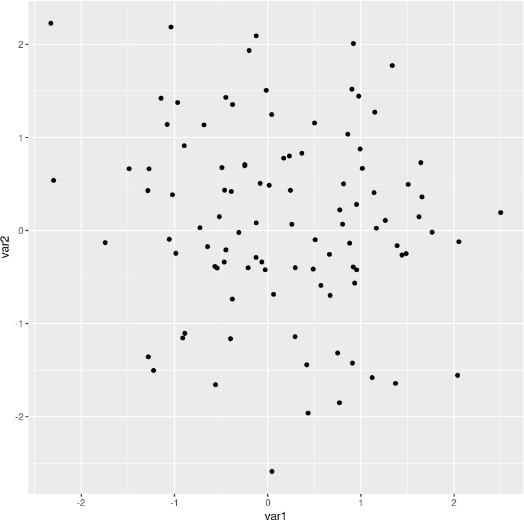
group = sample(1:4, 100, replace = TRUE)

)

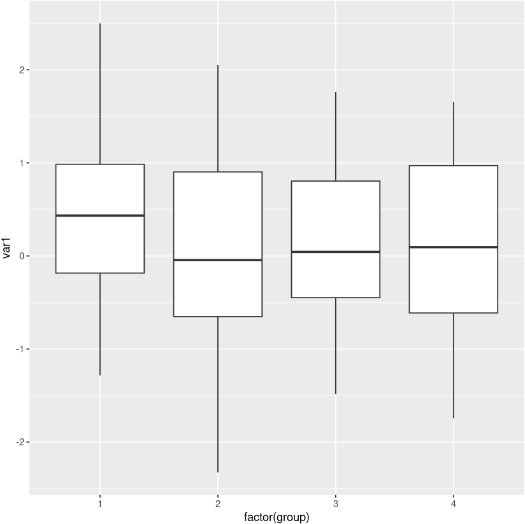
# Create a scatter plot

ggplot(data, aes(x = var1, y = var2)) + geom\_point()

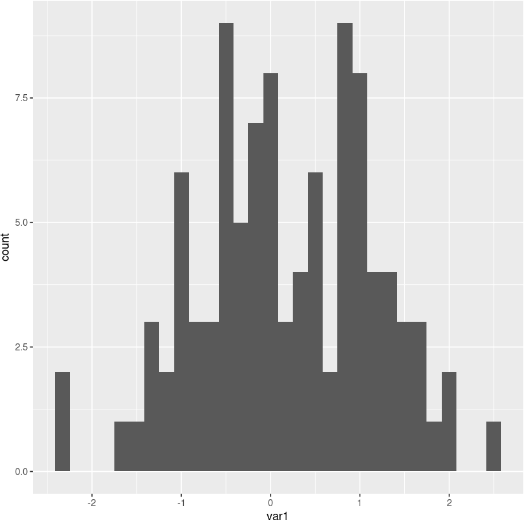
Output:



# Create a box plot

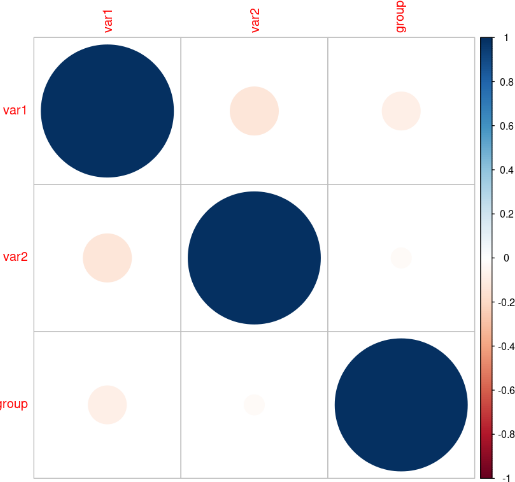
ggplot(data, aes(x = factor(group), y = var1)) + geom\_boxplot() Output:

# Create a histogram

ggplot(data, aes(x = var1)) + geom\_histogram() Output:

# Load the corrplot library library(corrplot)

# Create a correlation matrix plot corrplot(cor(data), method = "circle")



**Descriptive Statistical Measures** # Calculate the variance var(data$var1)

# Calculate the covariance cov(data$var1, data$var2) # Calculate the correlation cor(data$var1, data$var2) **Output:** 0.964993019401173

-0.131206113335423

-0.133108806509815

# Load the moments package library(moments)

# Load the psych library library(psych)

# Calculate the skewness skewness(data$var1)

# Calculate the kurtosis kurtosis(data$var1)

# Perform factor analysis fa(data)

#### Output:

-0.113671043634579

2.58907790883746

# Perform factor analysis fa(data)

#### Output:

Factor Analysis using method = minres Call: fa(r = data)

Standardized loadings (pattern matrix) based upon correlation matrix MR1 h2 u2 com

var1 1.00 0.9957 0.0043 1

var2 -0.13 0.0171 0.9829 1

group -0.08 0.0062 0.9938 1

MR1

SS loadings 1.02

Proportion Var 0.34

Mean item complexity = 1

Test of the hypothesis that 1 factor is sufficient.

df null model = 3 with the objective function = 0.03 with Chi Square = 2.53

df of the model are 0 and the objective function was 0

The root mean square of the residuals (RMSR) is 0.02 The df corrected root mean square of the residuals is NA

The harmonic n.obs is 100 with the empirical chi square 0.23 with prob < NA The total n.obs was 100 with Likelihood Chi Square = 0.12 with prob < NA

Tucker Lewis Index of factoring reliability = Inf Fit based upon off diagonal values = 0.95 Measures of factor score adequacy

MR1

Correlation of (regression) scores with factors 1.00 Multiple R square of scores with factors 1.00

Minimum correlation of possible factor scores 0.99

**PCA and LDA**

Two well-liked methods for multivariate analysis are PCA (Principal Component Analysis) and LDA (Linear Discriminant Analysis). Dimensionality reduction is accomplished with PCA, and classification is accomplished with LDA. For PCA and LDA in R, respectively, the lda() function from the MASS library and the prcomp() function from the stats package.

# Load the stats and MASS libraries library(stats)

library(MASS) # Perform PCA

pca <- prcomp(data[, 1:3]) summary(pca)

# Perform LDA

lda <- lda(group ~ var1 + var2, data = data) summary(lda)

Output:

Importance of components:

PC1 PC2 PC3

Standard deviation 1.0946 1.0498 0.9119

Proportion of Variance 0.3826 0.3519 0.2655

Cumulative Proportion 0.3826 0.7345 1.0000 Length Class Mode

prior 4

counts 4

means 8

scaling 4

lev 4

svd 2

N 1

call 3

terms 3

xlevels 0

-none- numeric

-none- numeric

-none- numeric

-none- numeric

-none- character

-none- numeric

-none- numeric

-none- call terms call

-none- list

#### RESAMPLING TECNIQUES

**Program**

**Creating a random permutation using R**

**set.seed**(4123)

score <‐ **c**(37,49,55,57,23,31,46)

perm <‐ **sample**(score, replace=F) perm

## [1] 46 55 37 31 23 57 49

Now, compute the mean of the first 4 entries, the last 3 and compute the difference mean.new <‐ **mean**(perm[1:4])

mean.new ## [1] 42.25

mean.trad <‐ **mean**(perm[5:7]) mean.trad

## [1] 43

diff <‐ mean.new‐mean.trad diff

## [1] ‐0.75

Another way is to create an index vector, and sample for just one group.

**set.seed**(4123)

N <‐ **length**(score)

index <‐ **sample**(N, size=4,replace=F) index

## [1] 7 3 1 6

score[index]

## [1] 46 55 37 31

score[‐index] ## [1] 49 57 23

diff2 <‐ **mean**(score[index])‐**mean**(score[‐index]) diff2

## [1] ‐0.75

##### Using R to generate many randomizations and compute p-value.

score <‐ **c**(37,49,55,57,23,31,46)

observed.diff <‐ **mean**(score[1:4] ‐ **mean**(score[5:7])) N <‐ **length**(score)

**set.seed**(4132) nperms <‐ 9999

perm.result <‐ **numeric**(nperms) *# vector to save the random differences*

for(i in 1:nperms)

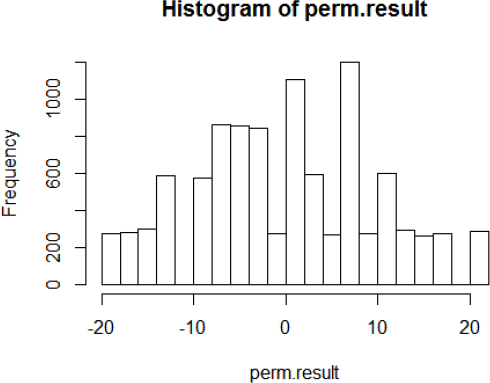
{

index <‐ **sample**(N, size=4, replace=FALSE) *#Choose 4 values w/o replacement*

perm.result[i] <‐ **mean**(score[index]) ‐ **mean**(score[‐index])

}

**hist**(perm.result)



(**sum**(perm.result >= observed.diff)+1)/(nperms + 1) *#P*‐*value*

## [1] 0.0562