Data Manipulation in R

Matt Arthur

UCR GradQuant

9 Nov 2021

Overview

Overview

Vector Operations

R is a *vectorized* language–meaning it is specially suited to processing lists and arrays of data. Let's create a numeric vector and do some summary measures on this vector:

Vector Operations

R is a *vectorized* language–meaning it is specially suited to processing lists and arrays of data. Let's create a numeric vector and do some summary measures on this vector:

```
x <- c(2, -3, 0, 8)
print(x)
```

```
## [1]  2 -3  0  8
```

```
cumsum(x)
```

```
## [1]  2 -1 -1  7
```

```
diff(x)
```

```
## [1] -5  3  8
```

Truncating and Rounding

Use `floor(x)` to return the nearest integer less than or equal to `x`:

```
x <- c(3.14159, 5, -1.223)
floor(x)

## [1]  3  5 -2
```

Use `ceiling(x)` to return the nearest integer greater than or equal to `x`:

```
ceiling(x)

## [1]  4  5 -1
```

Use `round(x)` to return the number nearest `x` defined by digits:

```
round(x, digits = 1)

## [1]  3.1  5.0 -1.2
```

Sorting and Ordering

R uses a variety of functions to sort, order, and rank numeric values:

Use `rev(x)` to reverse the order of elements in `x`:

```
x <- c(2, -3, 0, 8)
rev(x)

## [1]  8  0 -3  2
```

Use `sort(x)` to sort a vector into ascending/descending order:

```
sort(x, decreasing = TRUE)

## [1]  8  2  0 -3
```

Overview

Creating Strings in R

To define a string variable in R, simply pass a value encased in quotes (“ ”):

```
farm_animals <- c("pig","cow","sheep")
print(farm_animals)
```

```
## [1] "pig"   "cow"   "sheep"
```

Creating Strings in R

To define a string variable in R, simply pass a value encased in quotes (“ ”):

```
farm_animals <- c("pig","cow","sheep")
print(farm_animals)
```

```
## [1] "pig"   "cow"   "sheep"
```

Functions on Character Strings

The `toupper()` function replaces all lowercase letters with uppercase letters:

```
toupper(farm_animals)
```

```
## [1] "PIG"   "COW"   "SHEEP"
```

There is also a `tolower()` function that has the expected effect on strings.

Functions on Character Strings

`strsplit()` allows a string to be split into several components based on a delimiter:

```
dates <- c("10/01/2020","11/09/2020")
strsplit(dates, "/")
```

```
## [[1]]
## [1] "10"   "01"   "2020"
##
## [[2]]
## [1] "11"   "09"   "2020"
```

We can also concatenate strings using the `paste()` and `paste0()` functions:

```
paste("small", farm_animals, sep = " ")

## [1] "small pig"   "small cow"   "small sheep"

paste0("small ", farm_animals)

## [1] "small pig"   "small cow"   "small sheep"
```

Recycling is used with string concatenation in a similar fashion to numeric values:

```
paste(c("x", "y", "z"), c(1, 2), sep = "_")

## [1] "x_1" "y_2" "z_1"
```

Factors in R

R has a special way of storing character strings for non-numeric vectors.

```
grades <- c("A", "A", "B", "C", "A", "D", "B")
print(grades)

## [1] "A" "A" "B" "C" "A" "D" "B"

g_factors <- factor(grades,
                    levels = c("A","B","C","D","F"))
print(g_factors)

## [1] A A B C A D B
## Levels: A B C D F
```

Factors

Factors are a good way to store character data for two reasons:

1. Statistical models often need to know the levels of categorical variables.
2. Can be more efficient in terms of storage (especially in earlier versions of R)

Overview

Data Frames

Data Frames are R's way of storing *tabular* data: i.e., repeated measurements for related variables.

Data Frames

Data Frames are R's way of storing *tabular* data: i.e., repeated measurements for related variables.

Create data frames using the `data.frame()` function:

```
char <- c("R","S","T","U","V")
num <- 5:9
bool <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
frame <- data.frame(char, num, bool)
print(frame)
```

```
##   char num  bool
## 1    R   5  TRUE
## 2    S   6 FALSE
## 3    T   7  TRUE
## 4    U   8  TRUE
## 5    V   9 FALSE
```

Functions on Data Frames

R has several useful functions to extract elements from data frames.

- Extract row and column counts using `nrow()` and `ncol()`
- Elements that reside in cells of a data frame can be accessed using square-bracket indexing, similar to matrices.
- Extract entire columns from a data frame using `[[·]]`, `$`, or `[,column_name]`

Functions on Data Frames

R has several useful functions to extract elements from data frames.

- Extract row and column counts using `nrow()` and `ncol()`

```
nrow(frame)
## [1] 5
ncol(frame)
## [1] 3
```

Functions on Data Frames

R has several useful functions to extract elements from data frames.

- Elements that reside in cells of a data frame can be accessed using square-bracket indexing, similar to matrices.

```
frame[1,1]
## [1] R
## Levels: R S T U V
frame[1,]
##   char num bool
## 1    R   5 TRUE
frame[,1]
## [1] R S T U V
## Levels: R S T U V
```

Functions on Data Frames

R has several useful functions to extract elements from data frames.

- Extract entire columns from a data frame using `[[·]]`, `$`, or `[,column_name]`

```
frame[[1]]
## [1] R S T U V
## Levels: R S T U V
frame$char
## [1] R S T U V
## Levels: R S T U V
frame[,"char"]
## [1] R S T U V
## Levels: R S T U V
```

Overview

Exporting Data Files

R exports to a variety of different file types, including:

- Flat Files (CSV, General Delimited, etc)
- Excel
- SPSS
- STATA

```
print(frame)

##   char num  bool
## 1    R   5  TRUE
## 2    S   6 FALSE
## 3    T   7  TRUE
## 4    U   8  TRUE
## 5    V   9 FALSE
```

Exporting Data Files

R supports exports to a variety of different file types, including:

- Flat Files (CSV, General Delimited, etc)

```
write.table(frame, file = "~/Desktop/test.csv",
                   sep = ",", quote = F)
```

Exporting Data Files

R supports exports to a variety of different file types, including:

- Excel

```
library(writexl)
## Warning:  package 'writexl' was built under R
version 3.6.2
write_xlsx(frame, "~/Desktop/test.xlsx")
```

Exporting Data Files

R supports exports to a variety of different file types, including:

- SPSS

```
library(haven)
## Warning:  package 'haven' was built under R version
3.6.2
write_sav(frame, "test.sav")
```

Exporting Data Files

R supports exports to a variety of different file types, including:

- STATA

```
library(foreign)
write.dta(frame, "~/Desktop/test.dta")
```

Reading from Data Files

All the write-functions we have seen have corresponding read-functions to load data into an R session:

- Flat Files:

```
read.table(file = "~/Desktop/test.csv",
           header = T, sep = ",")
##   char num  bool
## 1    R   5  TRUE
## 2    S   6 FALSE
## 3    T   7  TRUE
## 4    U   8  TRUE
## 5    V   9 FALSE
```

Reading from Data Files

All the write-functions we have seen have corresponding read-functions to load data into an R session:

- Excel:

```
library(readxl)
read_xlsx(path = "~/Desktop/test.xlsx")
```

```
## # A tibble: 5 x 3
##   char    num bool
##   <chr> <dbl> <lgl>
## 1 R         5 TRUE
## 2 S         6 FALSE
## 3 T         7 TRUE
## 4 U         8 TRUE
## 5 V         9 FALSE
```

Reading from Data Files

All the write-functions we have seen have corresponding read-functions to load data into an R session:

- SPSS:

```
library(haven)
read_sav("test.sav")
```

```
## # A tibble: 5 x 3
##         char   num  bool
##   <dbl+lbl> <dbl> <dbl>
## 1     1 [R]     5     1
## 2     2 [S]     6     0
## 3     3 [T]     7     1
## 4     4 [U]     8     1
## 5     5 [V]     9     0
```

Reading from Data Files

All the write-functions we have seen have corresponding read-functions to load data into an R session:

- STATA:

```
library(foreign)
read.dta("~/Desktop/test.dta")
```

```
##   char num bool
## 1    R   5    1
## 2    S   6    0
## 3    T   7    1
## 4    U   8    1
## 5    V   9    0
```

Overview

Data Example

Example: evaluating wine quality based on physicochemical properties:

`https://archive.ics.uci.edu/ml/datasets/wine+quality`

Data Example

Example: evaluating wine quality based on physicochemical properties [1]

```
wine <- read.table("wine.csv", sep = ",", header = T)
head(wine)

##   fixed.acidity.volatile.acidity.citric.acid.residual.sugar
## 1
## 2
## 3
## 4
## 5
## 6
```

What happened here?

Data Example

Example: evaluating wine quality based on physicochemical properties

```
wine <- read.table("wine.csv", sep = ";", header = T)
wine <- wine[, c("quality",
                 "fixed.acidity",
                 "volatile.acidity")]
summary(wine)
```

```
##     quality        fixed.acidity   volatile.acidity
##  Min.   :3.000   Min.   : 4.60   Min.   :0.1200
##  1st Qu.:5.000   1st Qu.: 7.10   1st Qu.:0.3900
##  Median :6.000   Median : 7.90   Median :0.5200
##  Mean   :5.636   Mean   : 8.32   Mean   :0.5278
##  3rd Qu.:6.000   3rd Qu.: 9.20   3rd Qu.:0.6400
##  Max.   :8.000   Max.   :15.90   Max.   :1.5800
```

Data Example

Column names for the `wine` data frame:

Data Example

We can use the `unique()` to view the various values for `quality`:

```
unique(wine$quality)
```

```
## [1] 5 6 7 4 8 3
```

Data Example

We can reorder the rows in the `wine` data frame by using the `order()` function:

```
wineOrdered <- wine[order(wine$quality, decreasing = T), ]
head(wineOrdered)

##     quality fixed.acidity volatile.acidity
## 268       8           7.9             0.35
## 279       8          10.3             0.32
## 391       8           5.6             0.85
## 441       8          12.6             0.31
## 456       8          11.3             0.62
## 482       8           9.4             0.30
```

Data Example

Add new columns to the data frame directly, using the $ operator:

```
wine$average_acidity <- (1 / 2) *
                  (wine$fixed.acidity + wine$volatile.acidity)
head(wine$average_acidity)

## [1] 4.05 4.34 4.28 5.74 4.05 4.03
```

Data Example

Extract only rows where `quality = 8`:

```
wineHighQuality <- wine[wine$quality == 8, ]
head(wineHighQuality)
```

```
##     quality fixed.acidity volatile.acidity average_acidity
## 268       8           7.9             0.35           4.125
## 279       8          10.3             0.32           5.310
## 391       8           5.6             0.85           3.225
## 441       8          12.6             0.31           6.455
## 456       8          11.3             0.62           5.960
## 482       8           9.4             0.30           4.850
```

Data Example

Extract only rows where `quality = 8` **and** `fixed.acidity > 10`:

```
wineHighQuality2 <- wine[(wine$quality == 8 &
                          wine$fixed.acidity > 10), ]
head(wineHighQuality2)
```

```
##     quality fixed.acidity volatile.acidity average_acidity
## 279       8          10.3              0.32           5.310
## 441       8          12.6              0.31           6.455
## 456       8          11.3              0.62           5.960
## 496       8          10.7              0.35           5.525
## 499       8          10.7              0.35           5.525
```

Data Example

Extract only rows where `quality = 8` **or** `fixed.acidity > 10`:

```
wineHighQuality3 <- wine[(wine$quality == 8 |
                          wine$fixed.acidity > 10), ]
head(wineHighQuality3)
```

```
##     quality fixed.acidity volatile.acidity average_acidity
## 4         6          11.2              0.28           5.740
## 57        5          10.2              0.42           5.310
## 114       6          10.1              0.31           5.205
## 198       6          11.5              0.30           5.900
## 206       7          12.8              0.30           6.550
## 207       7          12.8              0.30           6.550
```

Data Example

Extract only rows where `quality` is equal to 3, 4, or 5:

```
wineLowerQuality <- wine[(wine$quality %in% c(3, 4, 5)), ]
head(wineLowerQuality)

##   quality fixed.acidity volatile.acidity average_acidity
## 1       5           7.4             0.70            4.05
## 2       5           7.8             0.88            4.34
## 3       5           7.8             0.76            4.28
## 5       5           7.4             0.70            4.05
## 6       5           7.4             0.66            4.03
## 7       5           7.9             0.60            4.25
```

Overview

Lists in R

Lists are general objects that are used for data storage and manipulation in R.

Lists in R

Lists are general objects that are used for data storage and manipulation in R.

- Lists are stored internally as *Generic Vectors*; i.e., vectors of generic objects

Lists in R

Lists are general objects that are used for data storage and manipulation in R.

- Lists are stored internally as *Generic Vectors*; i.e., vectors of generic objects
- Each element of a list can be a scalar, a vector, a function, or even another list!

Lists in R

Lists are general objects that are used for data storage and manipulation in R.

- Lists are stored internally as *Generic Vectors*; i.e., vectors of generic objects
- Each element of a list can be a scalar, a vector, a function, or even another list!
- Data Frames can be viewed as a special kind of list in R in which each element of the list is a column in the frame.

Creating a List in R

Lists are created from scratch using the `list()` function:

```
l <- list(numbers = c(1, 2, 3), letters = c("a", "b", "c"))
print(l)
```

```
## $numbers
## [1] 1 2 3
##
## $letters
## [1] "a" "b" "c"
```

Creating a List in R

Named elements of a list can be accessed exactly as columns in a data frame:

```
l$numbers

## [1] 1 2 3

l$letters

## [1] "a" "b" "c"
```

Creating a List in R

Named elements of a list can also be accessed using double square brackets:

```
l[[1]]

## [1] 1 2 3

l[[2]]

## [1] "a" "b" "c"
```

Lists and Data Frames

Note the relationship between lists and data frames:

```
class(l)

## [1] "list"

class(l) <- "data.frame"
attr(l, "row.names") <- 1:3
print(l)

##   numbers letters
## 1       1       a
## 2       2       b
## 3       3       c
```

The apply() Family

You may be familiar with the apply() function for matrices in R:

```
m <- matrix(1:9, nrow = 3)
print(m)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
apply(m, 2, sum) # column sums for m
```

```
## [1]  6 15 24
```

```
apply(m, 1, mean) # row means for m
```

```
## [1] 4 5 6
```

The `apply()` Family

R has a more generic function called `lapply(X, FUN, ...)` which applies the function `FUN` to each element of the list `X`.

The apply() Family

R has a more generic function called `lapply(X, FUN, ...)` which applies the function `FUN` to each element of the list `X`. Example: calculate e, e^2, and e^3:

```
lapply(1:3, exp)
```

```
## [[1]]
## [1] 2.718282
##
## [[2]]
## [1] 7.389056
##
## [[3]]
## [1] 20.08554
```

The `apply()` Family

`lapply()` also has more surprising uses. For instance, suppose we want to calculate a bootstrap confidence interval for the mean fixed acidity from our population.

The `apply()` Family

`lapply()` also has more surprising uses. For instance, suppose we want to write a function to calculate a bootstrap confidence interval for the mean fixed acidity from our population.

This would require:

1. Select M bootstrap samples–random subsamples of size n from our `wine` data, taken with replacement.
2. For the ith bootstrap sample, calculate a sample mean fixed acidity: $\bar{X}_i^*$
3. $\{\bar{X}_1^*, \ldots, \bar{X}_M^*\}$ constitutes a bootstrap sample that can be used to formulate a confidence interval for the true mean fixed acidity.

Can this be done without using a loop?

The apply() Family

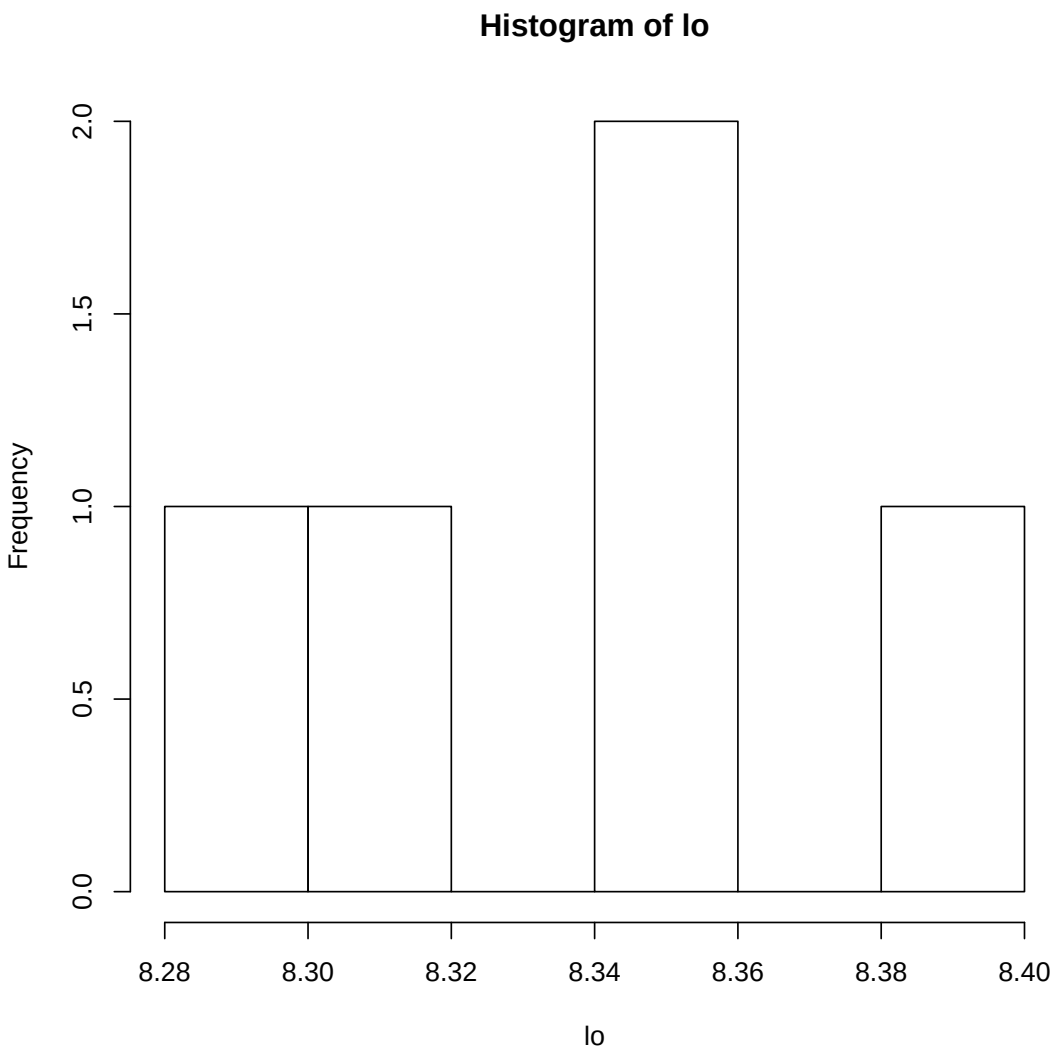
```
boot <- function(...) {
        n <- length(..2)
        y <- sample(..2, size = n, replace = T)
        return(mean(y))
}
draw <- function(m) {
        samples <- lapply(1:m, boot,
                          wine$fixed.acidity)
        class(samples) <- "numeric"
        return(samples)
}
```

The apply() Family

```
set.seed(123)
lo <- draw(5)
md <- draw(50)
hi <- draw(5000)
```

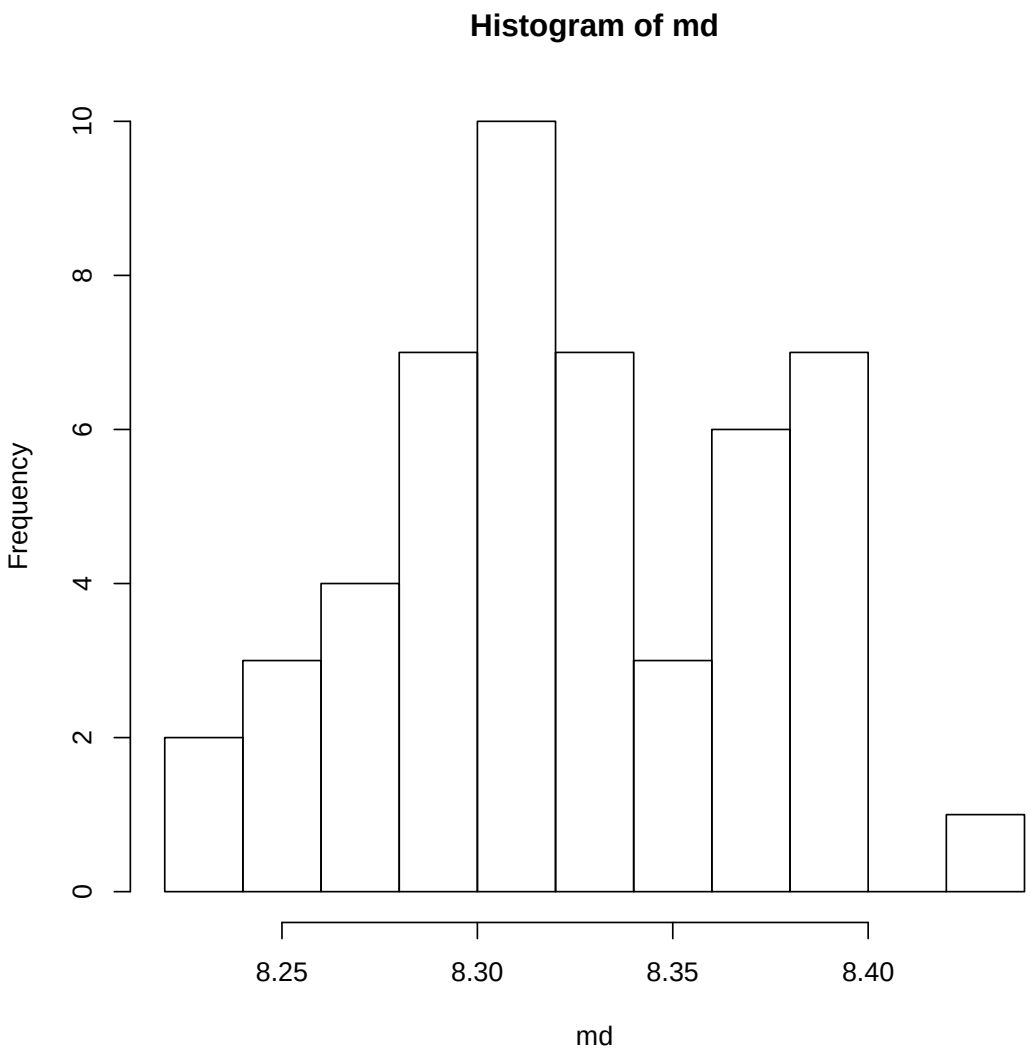
The apply() Family

```
hist(lo)
```



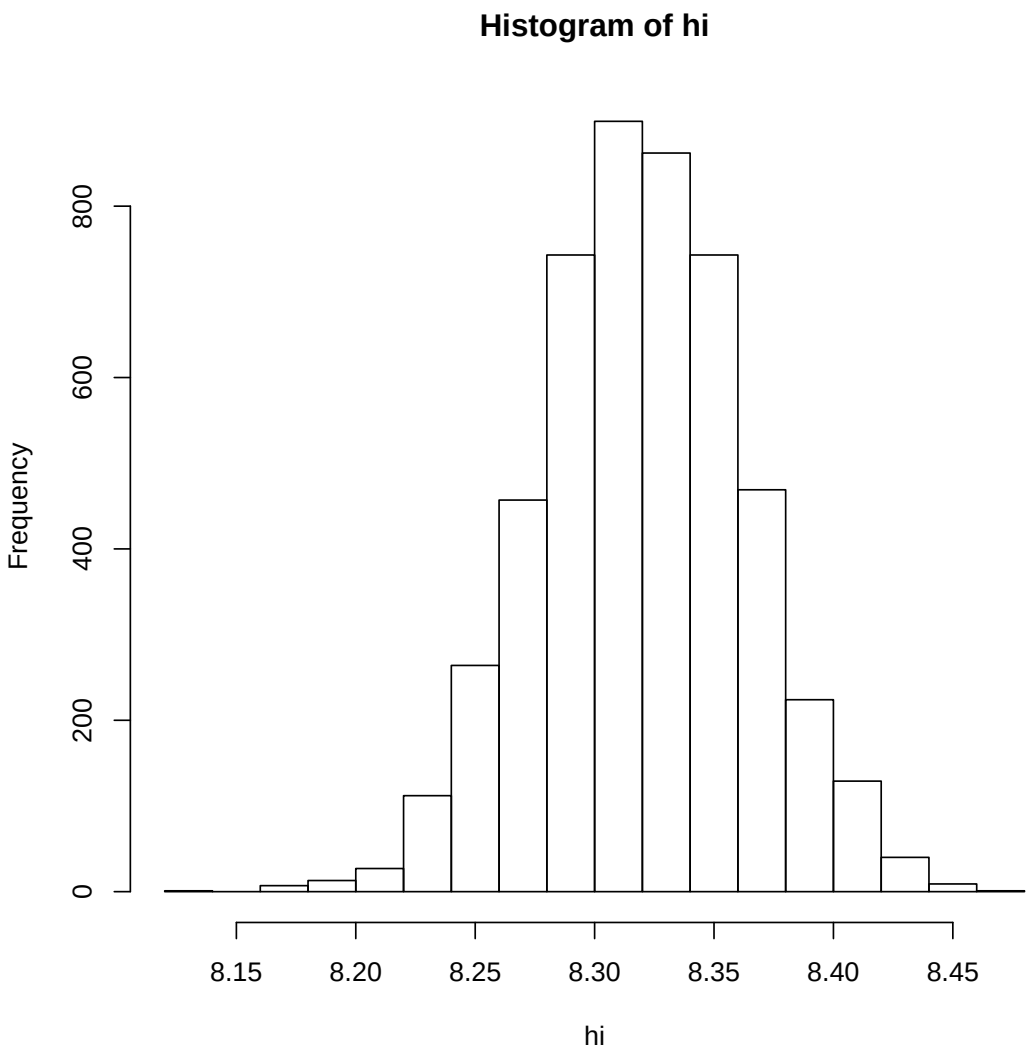
The apply() Family

```
hist(md)
```



The apply() Family

```
hist(hi)
```



The apply() Family

```
 lower <- 2 * mean(wine$fixed.acidity) -
          quantile(hi, 0.975)
 upper <- 2 * mean(wine$fixed.acidity) -
          quantile(hi, 0.025)
  names(lower) <- names(upper) <- NULL
  print(c(lo=lower, hi=upper))
```

```
##       lo       hi
## 8.233016 8.404819
```

Overview

Credits

P. Cortez, A. Cerdeira, F.Almeida, T. Matos, and J. Reis.
Modeling wine preferences by data mining from physicochemical properties.
In Decision Support Systems, Elsevier, 47(4):547–553, 2009.
https://archive.ics.uci.edu/ml/datasets/wine+quality.

Credits

This workshop was based in part on a similar workshop delivered by Ruihan Liu during Winter 2020.

Questions

The End.