Week 4 - Describing Data

Jesse Lecy

Monday, September 21, 2015

```
library(Lahman)
```

```
## Warning: package 'Lahman' was built under R version 3.1.3
```

```
data(Batting)
```

In this lecture, we want to explore some common functions that are used to **describe** our data. These functions are commonly used to create tables of descriptive statistics in research.

Describing Categorical Data

The most basic function for describing categorical data is the **table()** command, which counts the number of times each element occurs in a vector and reports it in a nice table.

```
t1 <- table( Batting$playerID )

head( t1 )
```

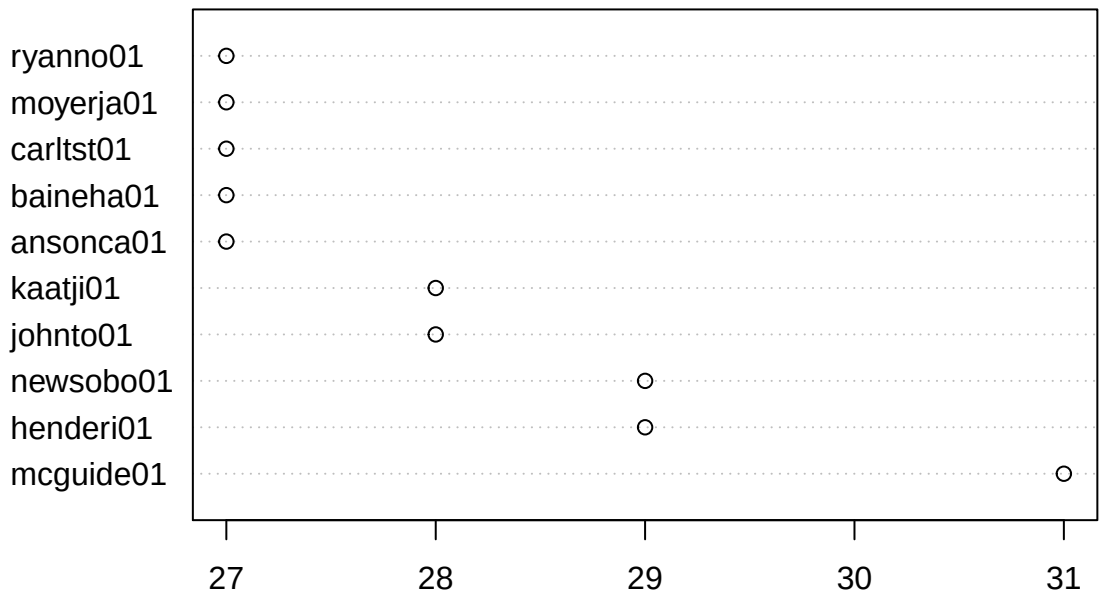
```
##
## aardsda01 aaronha01 aaronto01  aasedo01  abadan01  abadfe01
##         8        23         7        13         3         5
```

```
t1 <- sort( t1, decreasing=T )

# ten longest careers

t1[ 1:10 ]
```

```
##
## mcguide01 henderi01 newsobo01  johnto01  kaatji01 ansonca01 baineha01
##        31        29        29        28        28        27        27
## carltst01 moyerja01  ryanno01
##        27        27        27
```

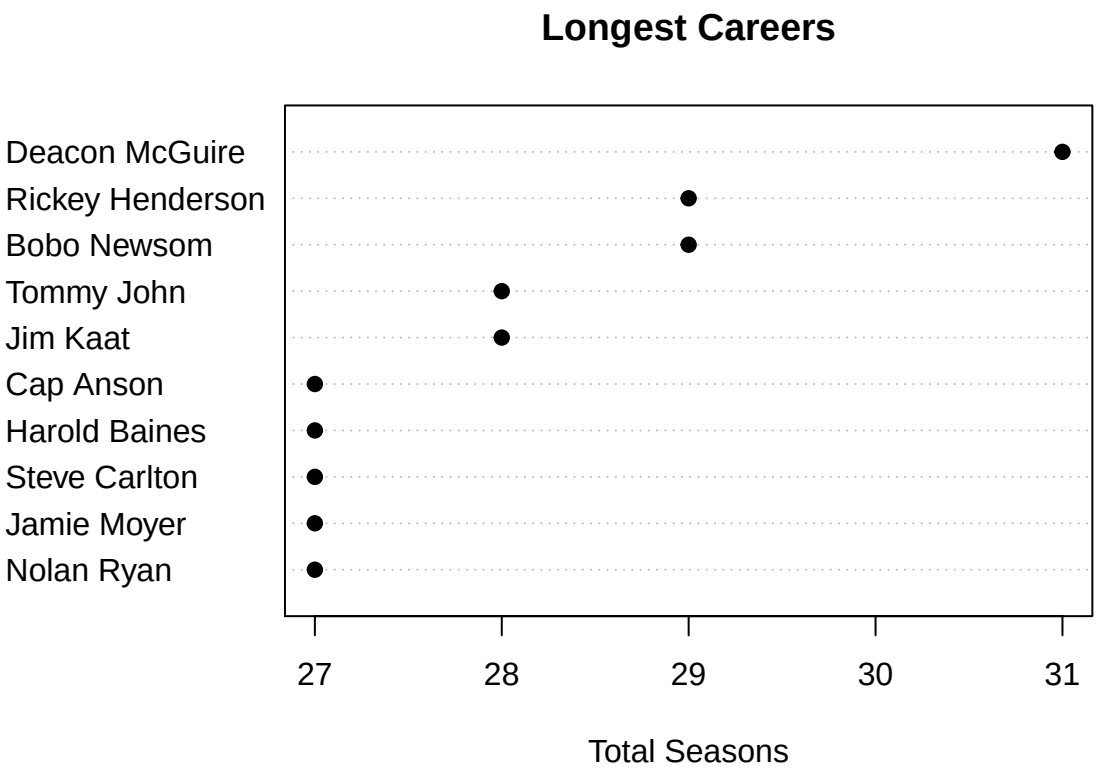
```
# In one line of code:
#
# sort( table( Batting$playerID ), decreasing=T )[ 1:10 ]

dotchart( t1[1:10] )
```

```
## Warning in dotchart(t1[1:10]): 'x' is neither a vector nor a matrix: using
## as.numeric(x)
```




```
# make pretty

player.name <- paste( Master$nameFirst, Master$nameLast )

name.match <- match( names(t1)[1:10], Master$playerID )

name.labels <- player.name[ name.match ]

dotchart( rev(t1[1:10]), labels=rev(name.labels), pch=19,
          xlab="Total Seasons", main="Longest Careers" )
```

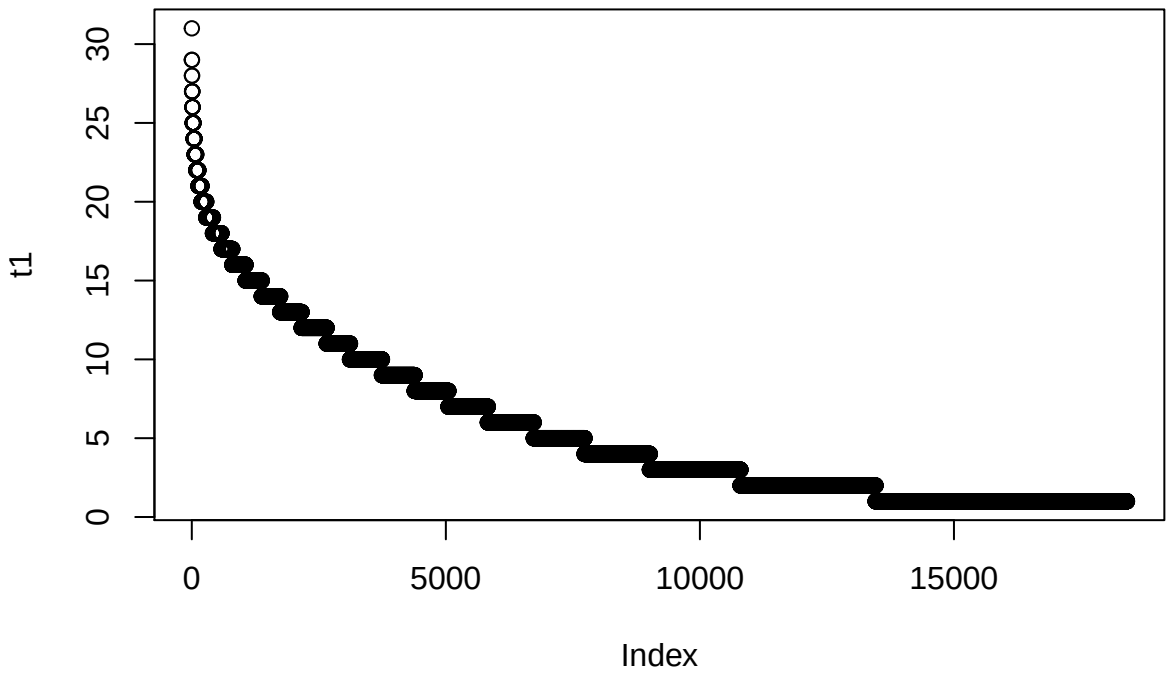
```
## Warning in dotchart(rev(t1[1:10]), labels = rev(name.labels), pch = 19, :
## 'x' is neither a vector nor a matrix: using as.numeric(x)
```



With categorical data, we may be more interested in the patterns than the specific individuals. Instead of asking, who has the longest career in baseball history, perhaps we want to know, how common is it to play more than 10 seasons?

It is not intuitive at first, but you can summarize data in a table with another **table()** function. In other words, the first table command counts the seasons for each player. The second counts the number of players that have played each amount.

```
t1 <- table( Batting$playerID )

t1 <- sort( t1, decreasing=T )

# what is the distribution of career spans?

plot( t1 )
```




```
t2 <- table( t1 )

t2  # number of players that have played in each season
```

```
## t1
##    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
## 4948 2659 1783 1286 1000  903  776  664  644  629  459  493  423  362  318
##   16   17   18   19   20   21   22   23   24   25   26   27   28   29   31
##  262  212  168  133   92   62   43   32   20   18    6    5    2    2    1
```

```
round( prop.table( t2 ), 2 )
```

```
## t1
##    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
## 0.27 0.14 0.10 0.07 0.05 0.05 0.04 0.04 0.03 0.03 0.02 0.03 0.02 0.02 0.02
##   16   17   18   19   20   21   22   23   24   25   26   27   28   29   31
## 0.01 0.01 0.01 0.01 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
```

```
# 27 percent of players make it only one season!

sum( prop.table(t2)[1:10] )
```

```
## [1] 0.8308612
```

```
# 17 percent of players make it past season 10
```

Describing Continuous Data

Using apply() and tapply()

If we want to generate some descriptive statistics on some quantitative variables, we can do this easily using functions like **mean()**, **median()**, **min()**, **max()**, **quantile()**, or to do them all at once, **summary()**.

So, if we are describing all of the data in one vector, our job is pretty easy.

Often times, though, we want to know about differences within our data. Do first basemen make more than second basemen? Does the American League have a higher batting average than the National League? Has pitching records changed over time?

In order to answer these questions, we could create complex subsets for each case (one subset of pitching records for each year from 1890 to present day, for example). But this is quite tedious. We need to learn how to **apply** the descriptive functions over the dataset using two or more variables at once in order to quickly summarize several variables, or look at summaries within groups.

Let's introduce some hypothetical data from an exeriment using a program that is meant to increase reading speed. For the experiment, the cases in the treatment group received instruction on how to increase reading speed, and practiced for a week. The control group received lollipops.

After the week, we want to test the subjects to see if their reading speeds differ. Since reading speed can be hard to measure, we give the subject five separate tests on five different texts, each lasting one minute. These different measures are averaged to get a more reliable measure of reading speed for each subject.

Our data looks like this:

```
read.speed
```

```
##    Measure1 Measure2 Measure3 Measure4 Measure5     group
## 1       232      255      252      257      241 treatment
## 2       250      237      222      250      258   control
## 3       240      249      238      249      238 treatment
## 4       218      225      215      223      212   control
## 5       236      238      260      236      237 treatment
## 6       257      279      247      252      249   control
## 7       285      298      279      303      284 treatment
## 8       236      241      264      252      250   control
## 9       231      231      218      234      216 treatment
## 10      196      208      210      186      210   control
```

Calculate Descriptives for Rows or Columns

If we want to generate descriptive statistics for our groups in this case, we would need to think in terms of individuals (rows), or tests (columns). Generating descriptives for all of these cases would require a lot of sub-sets.

R has a speedy way to do this using the **apply()** function. What we want to do is *apply* a descriptive statistic operation over all rows independently, or conversely all columns.

The apply function accepts three arguments. The first one (X) corresponds to the numeric variables that you would like to summarize. Note that we will drop the categorical variable for treatment group before using

apply(). The second argument, *MARGIN*, is used to specify whether we want the descriptives calculated for rows (MARGIN=1), or columns (MARGIN=2). And the third argument is where you specify the function that you would like to use to calculate your descriptive statistics.

```
# average reading speed of each person

apply( X=read.speed[,1:5], MARGIN=1, FUN=mean )
```

```
##     1     2     3     4     5     6     7     8     9    10
## 247.4 243.4 242.8 218.6 241.4 256.8 289.8 248.6 226.0 202.0
```

```
# were the tests equivalent, i.e. equally as hard?

apply( X=read.speed[,1:5], MARGIN=2, mean )
```

```
## Measure1 Measure2 Measure3 Measure4 Measure5
##    238.1    246.1    240.5    244.2    239.5
```

Note that if your descriptive function that you are applying requires additional arguments, you list those at the end and **apply()** will associate them with the function.

```
# if there are NAs present, add the na.rm argument

apply( X=read.speed[,1:5], MARGIN=1, mean, na.rm=T )
```

```
##     1     2     3     4     5     6     7     8     9    10
## 247.4 243.4 242.8 218.6 241.4 256.8 289.8 248.6 226.0 202.0
```

Calculate Descriptives for Groups

In most cases, our groups are not nicely packaged in rows and columns. Instead, they are often specified by categorical variables within our data. If we have one or two levels of these categorical variables then it is not too onerous to create subsets and calculate the statistics manually. But once the number of categories expands beyond a few the task becomes challenging.

The **tapply()** function calculates statistics over groups of data. In this case, X is a single numerical vector, and *INDEX* corresponds to your categorical variable.

Back to our reading speed experiment, let's see if reading speed has increased for our treatment group:

```
# did the treatment work?

tapply( X=read.speed$Measure5, INDEX=read.speed$group, FUN=mean )
```

```
##   control treatment
##     235.8     243.2
```

Unfortunately, our intervention does not seem to improve performance!

The **by()** function is similar to **tapply()** except it allows you to specific slightly more complex calculations. For examples, we can create a separate regression model for each group within our data to see if parameters change by population.

```
by( data=read.speed$Measure5, INDICES=read.speed$group, FUN=function(x){ mean(x) } )
```

```
## read.speed$group: control
## [1] 235.8
## ------------------------------------------------------------
## read.speed$group: treatment
## [1] 243.2
```

```
data( warpbreaks )
by( data=warpbreaks, INDICES=warpbreaks[,"tension"], FUN=function(x){ lm( breaks ~ wool, data = x) } )
```

```
## warpbreaks[, "tension"]: L
##
## Call:
## lm(formula = breaks ~ wool, data = x)
##
## Coefficients:
## (Intercept)        woolB
##       44.56       -16.33
##
## ------------------------------------------------------------
## warpbreaks[, "tension"]: M
##
## Call:
## lm(formula = breaks ~ wool, data = x)
##
## Coefficients:
## (Intercept)        woolB
##      24.000        4.778
##
## ------------------------------------------------------------
## warpbreaks[, "tension"]: H
##
## Call:
## lm(formula = breaks ~ wool, data = x)
##
## Coefficients:
## (Intercept)        woolB
##      24.556       -5.778
```

One more example using Lahman data:

```
data( Salaries )

sal.2002 <- Salaries[ Salaries$yearID == 2002 , ]

tapply( sal.2002$salary, sal.2002$teamID, sum, na.rm=T )
```

```
##       ANA       ARI       ATL       BAL       BOS       CAL       CHA
##  61721667 102819999  92870367  60493487 108366060        NA  57052833
##       CHN       CIN       CLE       COL       DET       FLO       HOU
```

```
## 75690833  45050390  78909449  56851043  55048000  41979917  63448417
##       KCA       LAA       LAN       MIA       MIL       MIN       ML4
##  47257000        NA  94850953        NA  50287833  40425000        NA
##       MON       NYA       NYM       NYN       OAK       PHI       PIT
##  38670500 125928583        NA  94633593  40004167  57954999  42323599
##       SDN       SEA       SFG       SFN       SLN       TBA       TEX
##  41425000  80282668        NA  78299835  74660875  34380000 105526122
##       TOR       WAS
##  76864333        NA
```

```
team.budget <- tapply( sal.2002$salary, sal.2002$teamID, sum, na.rm=T )

sort( format( team.budget, big.mark="," ), decreasing=T )
```

```
##           NYA           BOS           TEX           ARI           LAN
## "125,928,583" "108,366,060" "105,526,122" "102,819,999" " 94,850,953"
##           NYN           ATL           SEA           CLE           SFN
## " 94,633,593" " 92,870,367" " 80,282,668" " 78,909,449" " 78,299,835"
##           TOR           CHN           SLN           HOU           ANA
## " 76,864,333" " 75,690,833" " 74,660,875" " 63,448,417" " 61,721,667"
##           BAL           PHI           CHA           COL           DET
## " 60,493,487" " 57,954,999" " 57,052,833" " 56,851,043" " 55,048,000"
##           MIL           KCA           CIN           PIT           FLO
## " 50,287,833" " 47,257,000" " 45,050,390" " 42,323,599" " 41,979,917"
##           SDN           MIN           OAK           MON           TBA
## " 41,425,000" " 40,425,000" " 40,004,167" " 38,670,500" " 34,380,000"
##           CAL           LAA           MIA           ML4           NYM
## "          NA" "          NA" "          NA" "          NA" "          NA"
##           SFG           WAS
## "          NA" "          NA"
```

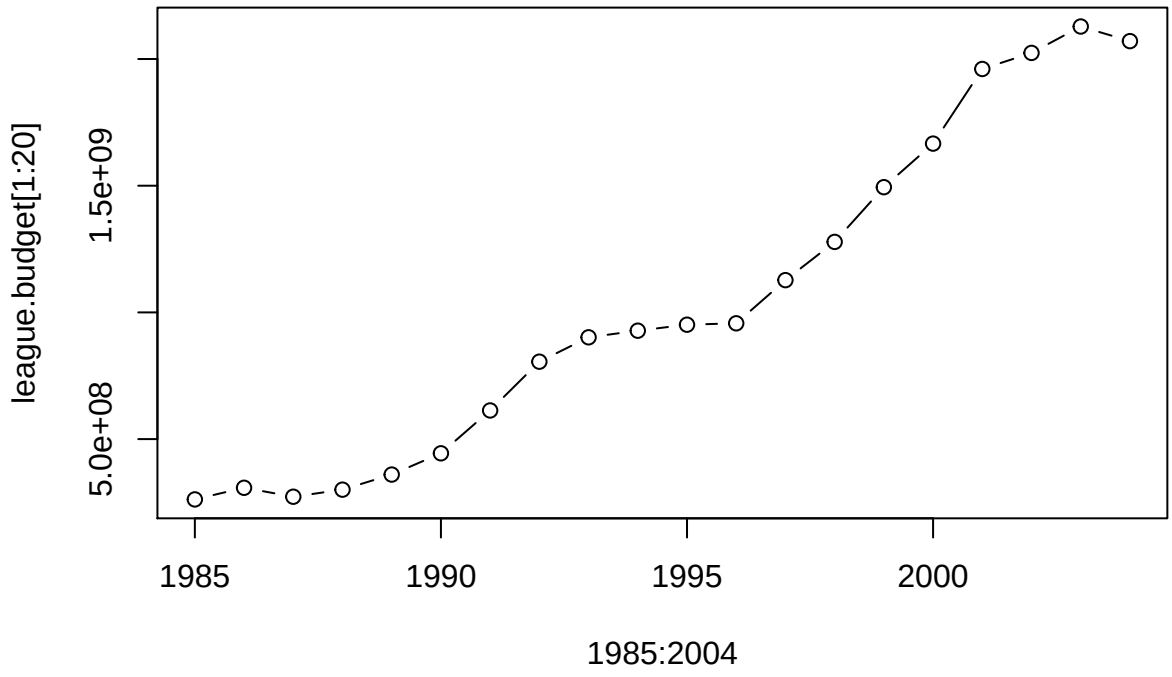
```
league.budget <- tapply( Salaries$salary, Salaries$yearID, sum, na.rm=T )
```

```
## Warning in FUN(X[[30L]], ...): integer overflow - use sum(as.numeric(.))

## Warning in FUN(X[[30L]], ...): integer overflow - use sum(as.numeric(.))

## Warning in FUN(X[[30L]], ...): integer overflow - use sum(as.numeric(.))

## Warning in FUN(X[[30L]], ...): integer overflow - use sum(as.numeric(.))

## Warning in FUN(X[[30L]], ...): integer overflow - use sum(as.numeric(.))

## Warning in FUN(X[[30L]], ...): integer overflow - use sum(as.numeric(.))

## Warning in FUN(X[[30L]], ...): integer overflow - use sum(as.numeric(.))

## Warning in FUN(X[[30L]], ...): integer overflow - use sum(as.numeric(.))

## Warning in FUN(X[[30L]], ...): integer overflow - use sum(as.numeric(.))

## Warning in FUN(X[[30L]], ...): integer overflow - use sum(as.numeric(.))
```

```
format( league.budget, big.mark="," )
```

```
##            1985            1986            1987            1988
## "  261,964,696" "  307,854,518" "  272,575,375" "  300,452,424"
##            1989            1990            1991            1992
## "  359,995,711" "  443,881,193" "  613,048,418" "  805,543,323"
##            1993            1994            1995            1996
## "  901,740,134" "  927,836,287" "  951,469,367" "  956,983,550"
##            1997            1998            1999            2000
## "1,127,285,885" "1,278,282,871" "1,494,228,750" "1,666,135,102"
##            2001            2002            2003            2004
## "1,960,663,313" "2,024,077,522" "2,128,262,128" "2,070,665,943"
##            2005            2006            2007            2008
## "           NA" "           NA" "           NA" "           NA"
##            2009            2010            2011            2012
## "           NA" "           NA" "           NA" "           NA"
##            2013            2014
## "           NA" "           NA"
```

```
plot( 1985:2004, league.budget[1:20], type="b" )
```



Note that we can also create groups using logical statements or the **cut()** function (see below).

Assigning Continuous Data to Categories

There are many instances where we want to look at descriptive statistics for one variable, and examine how it varies over different levels of another quantitative variable. For example, perhaps we want to see the percentage of households insured for the bottom, middle, and upper income groups.

In order to transform a numerical measure to a categorical one, we use the **cut()** function. This function uses an argument called *breaks* to determine how to split the continuous variable into groups. The argument either accepts a single numeric value, in which case it creates that number of groups using an equal intervals argument. Or else you can tell it where the breaks should be in your data.

The *labels* argument allows you to name the groups. The default is to use the break points.

The result will be a categorical (factor) vector where each element has been assigned to its proper group.

```
x <- sample( 1:9, 9 )

x
```

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

```
cut( x, breaks=3 )
```

```
## [1] (6.33,9.01]  (0.992,3.67] (3.67,6.33]  (6.33,9.01]  (0.992,3.67]
## [6] (3.67,6.33]  (0.992,3.67] (6.33,9.01]  (3.67,6.33]
## Levels: (0.992,3.67] (3.67,6.33] (6.33,9.01]
```

```
cut( x, breaks=3, labels=c("low","middle","high") )
```

```
## [1] high   low    middle high   low    middle low    high   middle
## Levels: low middle high
```

```
cut( x, breaks=c(0,2.5,7.5,10) )  # specify the break points
```

```
## [1] (2.5,7.5] (0,2.5]   (2.5,7.5] (7.5,10]  (0,2.5]   (2.5,7.5] (2.5,7.5]
## [8] (7.5,10]  (2.5,7.5]
## Levels: (0,2.5] (2.5,7.5] (7.5,10]
```

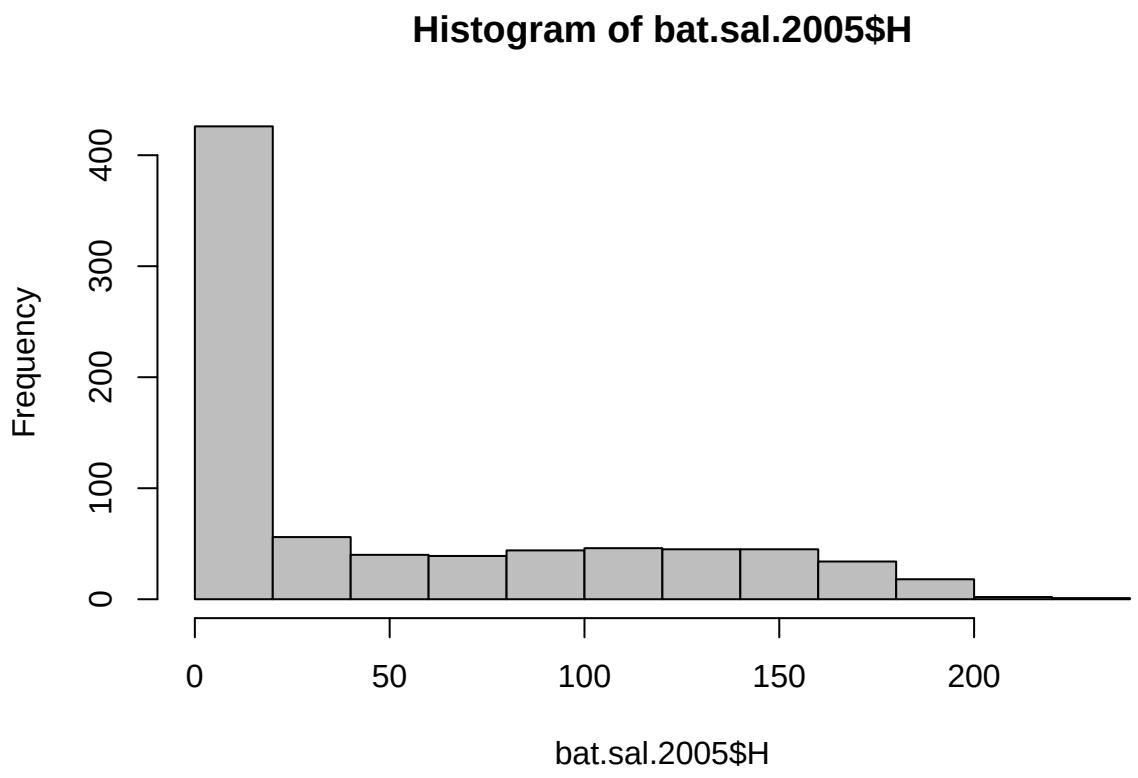
```
# 0->  1,2  <-2.5->  3,4,5,6,7  <-7.5->  8,9  <-10
```

An example using Lahman data:

```
data( Batting )
data( Salaries )

bat.2005 <- Batting[ Batting$yearID == 2005 , c("playerID","yearID","teamID","H","SO","HR") ]

bat.sal.2005 <- merge( bat.2005, Salaries )

summary( bat.sal.2005$H )
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    0.00   11.00   48.91   99.00  221.00
```

```
hist( bat.sal.2005$H, col="gray" )
```




```
hit.lev <- cut( bat.sal.2005$H, breaks=5 )

table( hit.lev )
```

```
## hit.lev
## (-0.221,44.2]   (44.2,88.4]    (88.4,133]     (133,177]     (177,221]
##           491            83           107            92            23
```

```
tapply( bat.sal.2005$H, hit.lev, mean, na.rm=T )
```

```
## (-0.221,44.2]   (44.2,88.4]    (88.4,133]     (133,177]     (177,221]
##      6.258656     65.602410    110.102804    154.532609    192.173913
```

```
hit.rank.lev <- cut( rank(bat.sal.2005$H), breaks=5 )

table( hit.rank.lev )
```

```
## hit.rank.lev
## (126,261] (261,395] (395,528] (528,662] (662,797]
##       253       144       131       133       135
```

```
tapply( bat.sal.2005$H, hit.rank.lev, mean, na.rm=T )
```

```
##  (126,261]  (261,395]  (395,528]  (528,662]  (662,797]
##   0.000000   4.222222  33.870229  95.616541 156.844444
```