[187]:
```
## Importing packages

# This R environment comes with all of CRAN and many other helpful packag
# You can see which packages are installed by checking out the kaggle/rst
# https://github.com/kaggle/docker-rstats

library(tidyverse) # metapackage with lots of helpful functions

## Running code

# In a notebook, you can run a single code cell by clicking in the cell a
# the blue arrow to the left, or by clicking in the cell and pressing Shi
# you can run code by highlighting the code you want to run and then clic
# at the bottom of this window.

## Reading in files

# You can access files from datasets you've added to this kernel in the '
# You can see the files added to this kernel by running the code below.

list.files(path = "../input")

## Saving data

# If you save any files or images, these will be put in the "output" dire
# can see the output directory by committing and running your kernel (usi
# Commit & Run button) and then checking out the compiled version of your
```

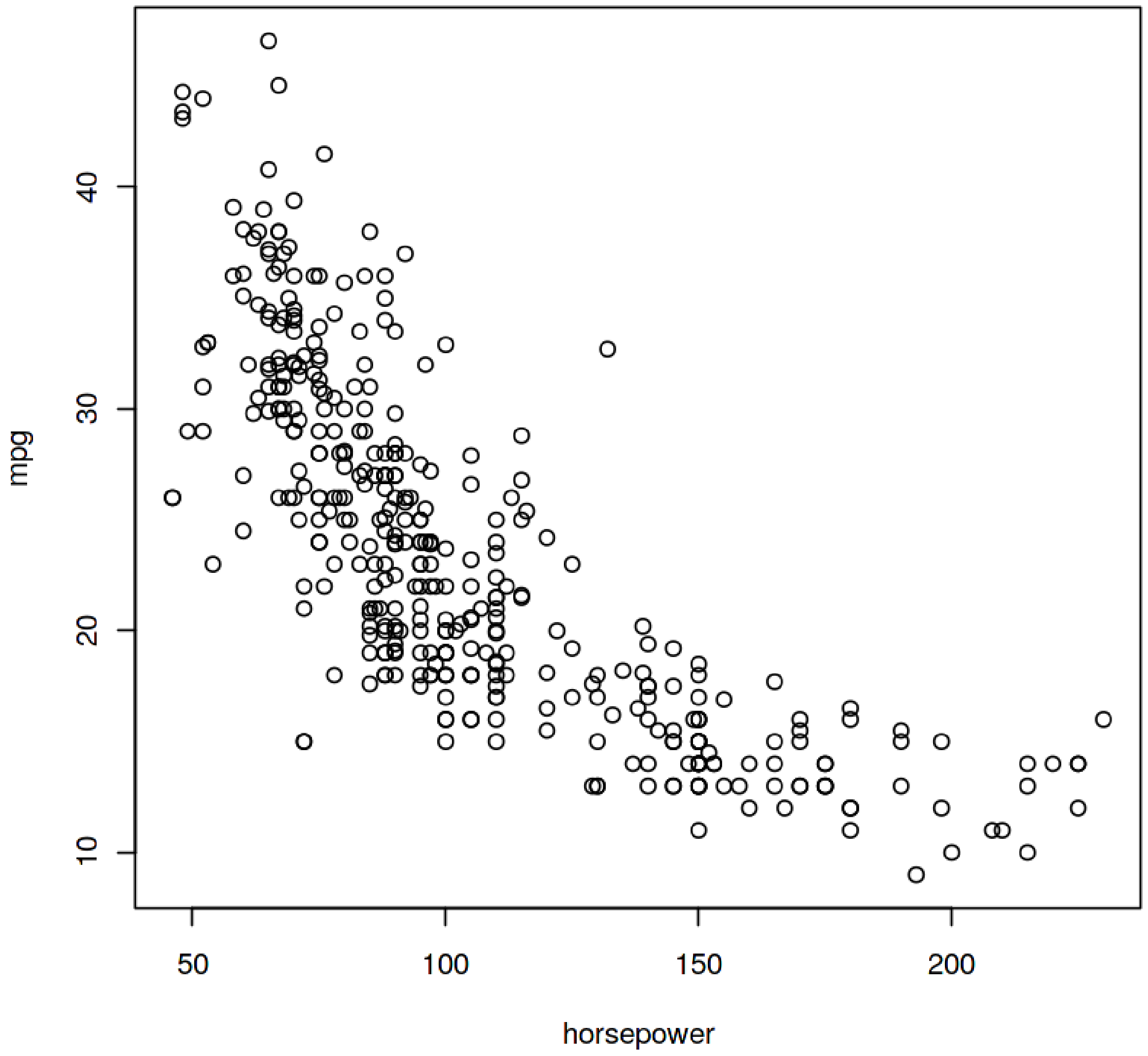
[188]:
```
#
#  https://youtu.be/6dSXlqHAoMk
#
```

[189]:
```
# requires, just like using library, it will also return a true
# and false if the package doesn't exist.
require(ISLR) # our data for our session

require(boot)
```

[190]:
```
# So that's a general cross-validation package for glms.
#?cv.glm
```

[191]:

```
# So we're gonna use the auto data. And in particular,
# we look at two variables, miles per gallon and horsepower.
plot(mpg~horsepower,data=Auto)
# And we see, as we might expect, miles per gallon drops down
# quite substantially as horsepower increases.
```



[192]:

```
## LOOCV
```

[193]:

```
# And so we'll fit a linear model.
# And we'll use glm to fit this, even though we just fit in a linear mode
# So glm can fit nonlinear models as well, in particular logistic regress
# But it will also fit linear models.
glm.fit=glm(mpg~horsepower, data=Auto)
cv.glm(Auto,glm.fit)$delta
# for cv.glm, the default is to set K equal to the number of observations
# which gives the usual leave-one-out cross-validation.
#
#cv.glm actually does LOOCV by brute force, it actually refits the model
#pretty slow (doesnt use formula (5.2) on page 180)
#
# And eventually it came up and produced two numbers.
# (Well, it produced quite a lot actually. But we just looked at the delt
# which is the cross-validated prediction error.)
#
# The first number is the raw leave-one-out, or lieu cross-validation res
# And the second one is a bias-corrected version of it.
# (the bias correction has to do with the fact that the data
# set that we train it on is slightly smaller than the one
# that we actually would like to get the error for, which is
# the full data set of size n.
# Turns out that has more of an effect for k-fold cross-validation.)
#
```

```
24.2315135179293   24.2311440937562
```

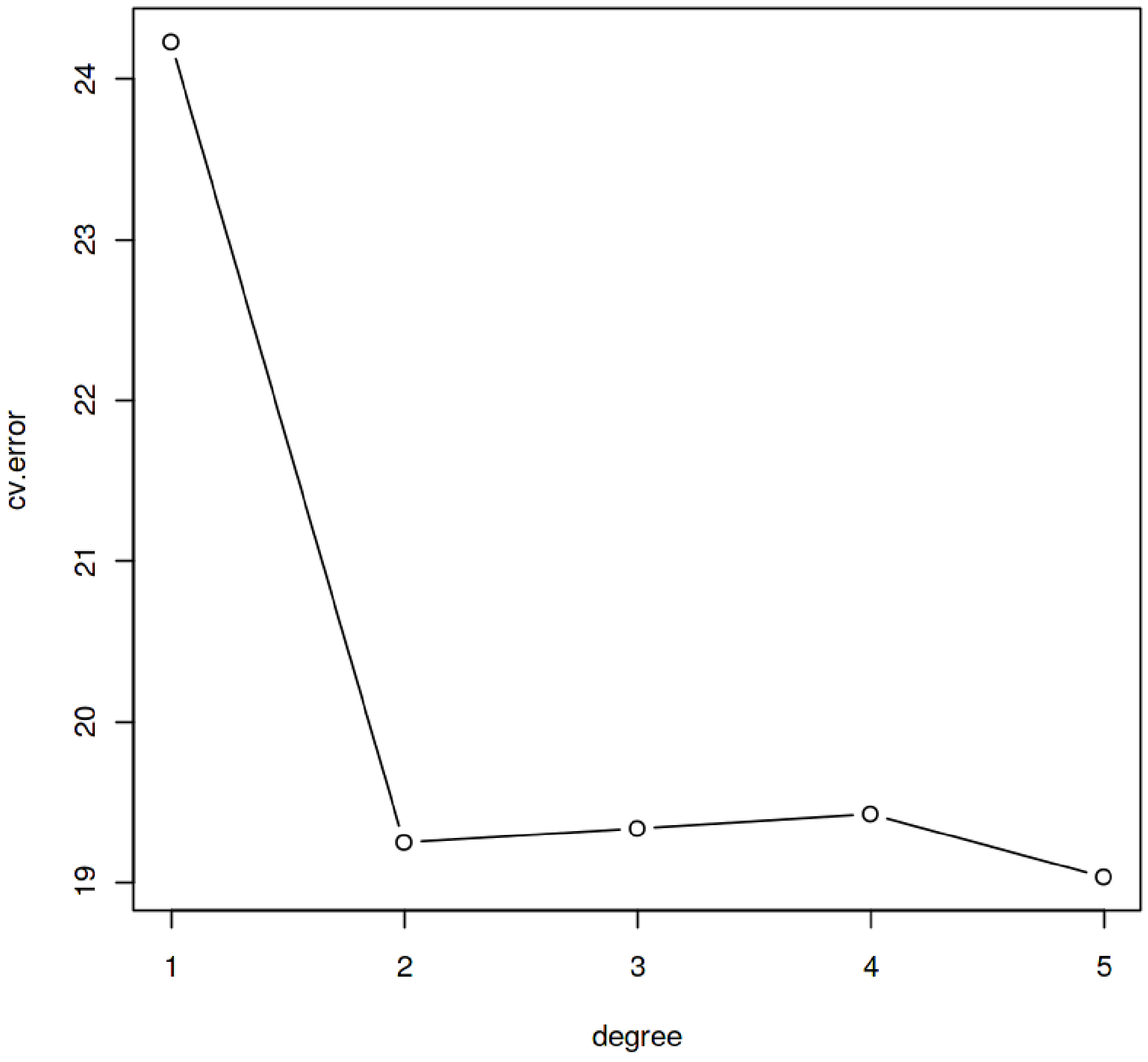
[194]:
```
# Now the thing is for leave-one-out cross-validation and for linear mode
# , this function doesn't exploit the nice simple formula we saw in the c
# what is this nice formula ?
# "leave-one-out sum of squared errors." = (5.1) = (5.2)
#                                               sum{[(yi-yi_hat)/(1-Hii)]^
#  Lets write a simple function to use formula (5.2)
loocv=function(fit){
  h=lm.influence(fit)$h # to put that in a vector h
  mean((residuals(fit)/(1-h))^2)
  # the residue of fit and 1-h are vectors,
  # it devides that element by element
}

# The Hii (h here in the formula, vart between 0 and 1) that we have ther
# is the diagonal element of the hat matrix.
# The hat matrix is the operator matrix that produces the least squares f
# This is also known as the self influence.
# It's a measure of how much observation i contributes to it's own fit.
# And if Hii is close to 1, in other words observation i
# really contributes a lot to its own fit, 1 minus Hii is small.
# And that will inflate that particular residual.
# So this is like a magic formula.
# It tells you that you can get your cross-validated fit by
# the simple modification of the residuals from the full fit.
# And that's much more efficient, and cheaper to compute.
```

[195]:
```
## Now we try it out
loocv(glm.fit)

# very quickly it produced the 24.23,
# that we saw above for the first element of the results of cv.glm.
```

```
24.2315135179292
```

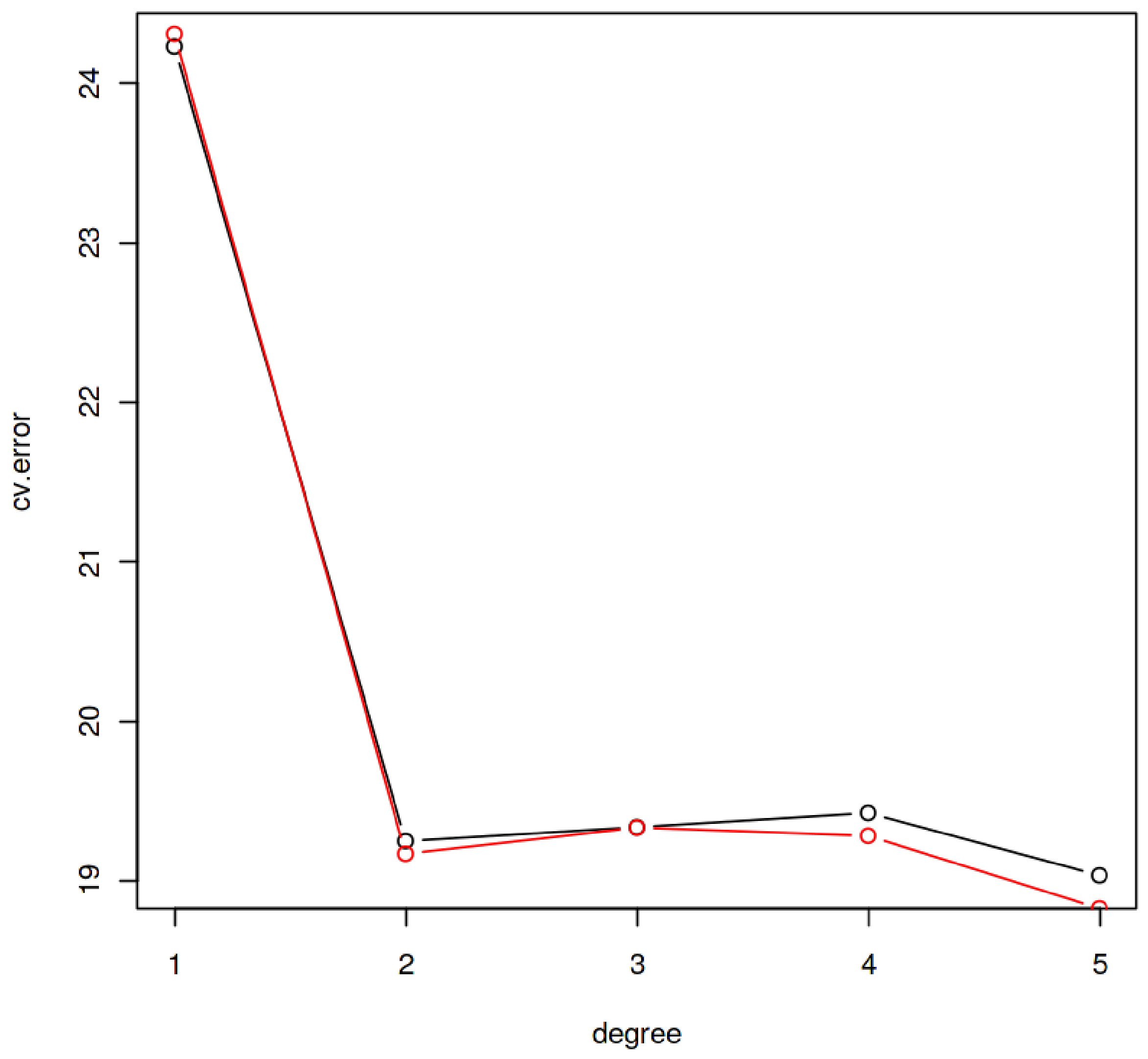
[196]:

```
# And now we're going to fit some polynomials of degrees 1 up to 5.
cv.error=rep(0,5) # a vector for collecting the errors
# replicates 0, 5 times in cv.error => cv.error = 0 0 0 0 0
degree=1:5 # degree = 1 2 3 4 5
for(d in degree){
  glm.fit=glm(mpg~poly(horsepower,d), data=Auto)
  # we use the poly function, the function of horsepower and degree.
  cv.error[d]=loocv(glm.fit)
}
plot(degree,cv.error,type="b")
```



[197]:

```
# Let's try 10-fold cross-validation out.

# So for 10-fall cross-validation, you only have to fit the model 10 time
# With leave-one-out you have to in principle fit the model n
# times, where n is the number of training points.
# Although we did have the shortcut for linear regression.
# The reason cv.glm doesn't use that shortcut is that it's
# also set up to work on logistic regressions and other
# models, and there the shortcut doesn't work.

## 10-fold CV
cv.error10=rep(0,5)
for(d in degree){
  glm.fit=glm(mpg~poly(horsepower,d), data=Auto)
  cv.error10[d]=cv.glm(Auto,glm.fit,K=10)$delta[1]
  # And now we'll actually use a cv.glm function to compute the errors.
  # And so we call cv.glm, and we tell it k is 10.
  # So that tells the number of folds.
}
plot(degree,cv.error,type="b")
# need to keep this otherwise error message showed up :
# Error in plot.xy(xy.coords(x, y), type = type, ...):
# plot.new has not been called yet
# Traceback:
# 1. lines(degree, cv.error10, type = "b", col = "red")
# 2. lines.default(degree, cv.error10, type = "b", col = "red")
# 3. plot.xy(xy.coords(x, y), type = type, ...)
lines(degree,cv.error10,type="b",col="red")
```



[198]:

```
# In general we favor 10-fold cross-validation for computing errors.
# It tends to be a more stable measure than leave-one-out cross-validatio
# And for the most time, it's cheaper to compute.
```

[199]:

```
## Bootstrap
# https://youtu.be/YVSmsWoBKnA
#
```

[200]:

```
# The bootstrap is one of the really powerful tools we have
# And what it does is it lets you get at the sampling distribution of sta
# for which it's really hard to develop theoretical versions.
# So the bootstrap gives us a really easy way of doing
# statistics when the theory is very hard.
```

[201]:

```
## Minimum risk investment - Section 5.2
```

[202]:

```
alpha=function(x,y){
  vx=var(x)
  vy=var(y)
  cxy=cov(x,y)
  (vy-cxy)/(vx+vy-2*cxy)
}
# closed parentheses, which means that the function will return
# the last line that was evaluated, which is actually our alpha.

alpha(Portfolio$X,Portfolio$Y)
```

```
0.57583207459283
```

[203]:

```
# test Portfolio
Portfolio[1:5,]
```

| X | Y |
|---|---|
| -0.8952509 | -0.2349235 |
| -1.5624543 | -0.8851760 |
| -0.4170899 | 0.2718880 |
| 1.0443557 | -0.7341975 |
| -0.3155684 | 0.8419834 |

[204]:
```
# what is the sampling variability of alpha?
# (What's the standard error of alpha?
# How variable is it going to be?)
#
# that's a non-linear formula of x and y.
# And we just wouldn't know a priori how to do that.
# This is a case where the bootstrap really helps out.
```

[205]:
```
# in order to use the bootstrap function we need to
# make a little wrapper that allows a bootstrap to work.
alpha.fn=function(data, index){   # data: we take a data frame, index: r
  with(data[index,],alpha(X,Y))
}

# we take a data (= data frame), and index (= row of data frame), and com
# (in this case, alpha index, for which you want to compute the variance
# index has values 1 to n, and there will be n of them.
# it uses the function "with", which is a very handy function,
# "With" takes first argument of data frame and then some commands.
# And what it says is, using the data in the data frame, execute the comm
# So in this case, we use with data of index,
# so that gets the right observations for this particular bootstrap sampl
# Compute alpha of x and y.
```

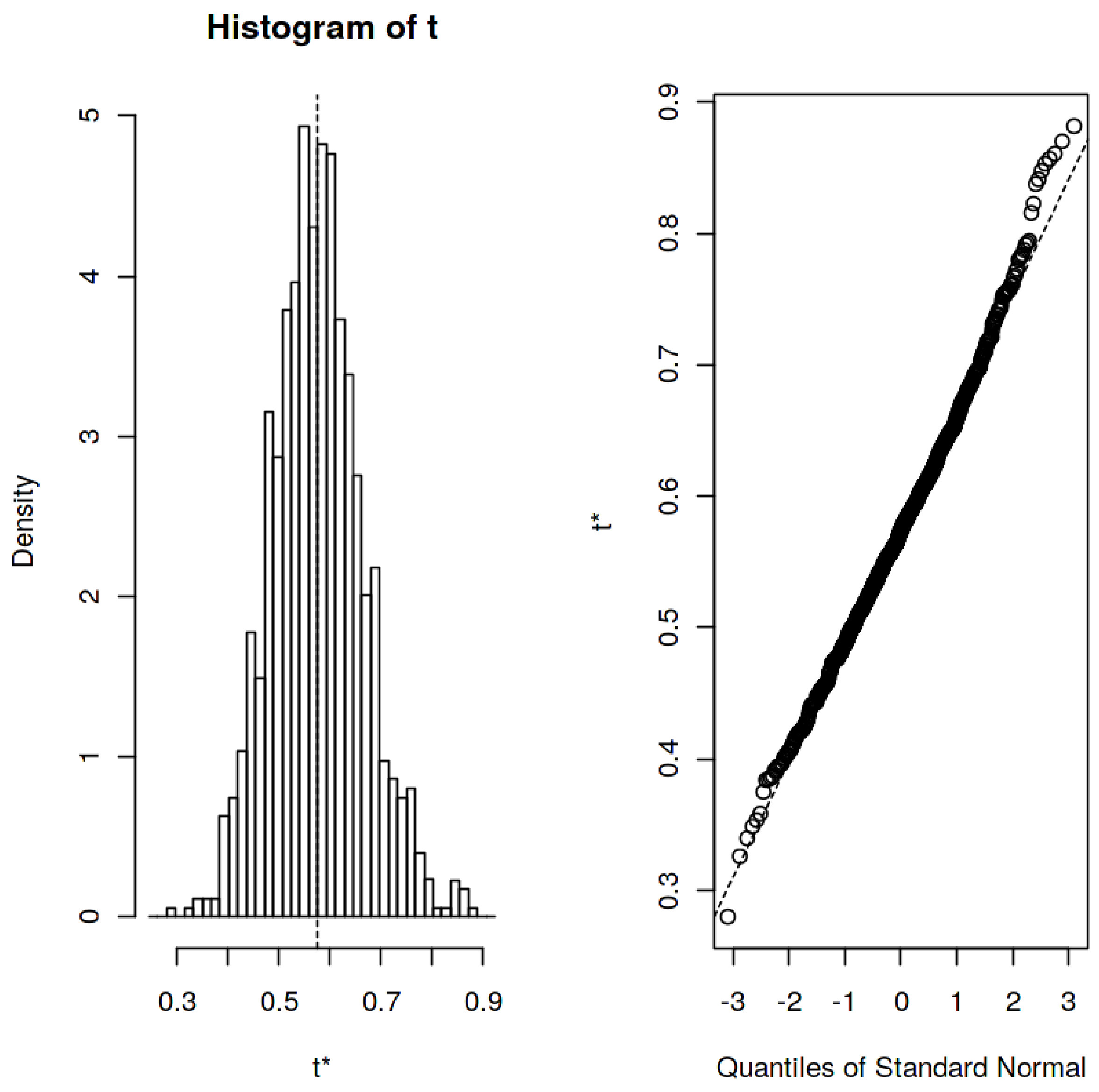
[206]:
```
#alpha.fn(Portfolio,1:100)

# we can get the same value as what we get before, okay it works !
# let do bootstrap next
```

[207]:
```
# let's do bootstrap.
#
# And since a bootstrap involves random sampling,
# and if we want to get reproducible results just for purpose of demonstr
# it's good to set the random number seed. So there we set seed 1.
set.seed(1)

alpha.fn (Portfolio,sample(1:100,100,replace=TRUE))
# we take a random sample instead of giving an index 1 to n.
# So here we've sampled the numbers 1 to 100, sample of size 100,
# with replace equals to true.
# This is the kind of thing the bootstrap's going to do over and over.
# Here we just do it once.
```

```
0.596383302006392
```

[208]:
```
boot.out=boot(Portfolio,alpha.fn,R=1000)
# do 1000 bootstrap

boot.out
# it tells us our original statistic was 0.575, and it gives us the estim
# of bias and standard error. We were interested in the standard error.
# The bias is negligible. The standard error in this case is 0.08.
#

plot(boot.out)
# And you get a two plots. One is a histogram and it looks like a pretty
# maybe Gaussian.
# And the second plot is a qqplot, which plots the ordered values against
# And if it lines up on a straight line like it pretty much does here,
# you may say it looks close to Gaussian, maybe a slightly bigger tail or

#?boot
```

```
ORDINARY NONPARAMETRIC BOOTSTRAP


Call:
boot(data = Portfolio, statistic = alpha.fn, R = 1000)


Bootstrap Statistics :
     original        bias    std. error
t1* 0.5758321 -7.315422e-05  0.08861826
```



[209]:

```
# test sample:
set.seed(1)
sample(1:100,100,replace=TRUE)
```

```
27  38  58  91  21  90  95  67  63  7   21  18  69  39  77  50  72  100  39  78  94
22  66  13  27  39  2   39  87  35  49  60  50  19  83  67  80  11  73   42  83  65
79  56  53  79  3   48  74  70  48  87  44  25  8   10  32  52  67  41   92  30  46
34  66  26  48  77  9   88  34  84  35  34  48  90  87  39  78  97  44   72  40  33
76  21  72  13  25  15  24  6   65  88  78  80  46  42  82  61
```

```
[ ]:
```