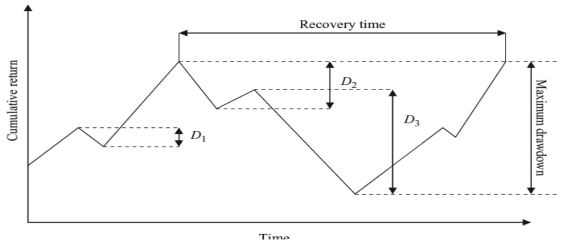
Stat 107: Introduction to Business and Financial Statistics
Class 6: Density Estimation and the Bootstrap

Drawdown

- What if you were a horrible investor?
- You always bought at the maximum price and sold at the minimum price.
- That would be the worst case scenario for any stock.
- Drawdown measures that over time.

Drawdown

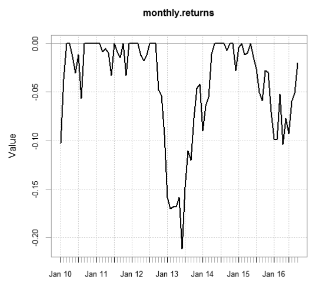


Investopedia explains *Drawdown*
A drawdown is measured from the time a retrenchment begins to when a new high is reached. This method is used because a valley can't be measured until a new high occurs. Once the new high is reached, the percentage change from the old high to the smallest trough is recorded.

PerformanceAnalytics in R

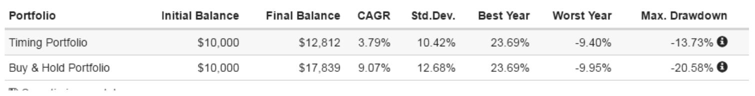
```
require(quantmod)
require(PerformanceAnalytics)
getSymbols("AAPL",from="2010-01-01")
#  calculate returns based on Adjusted Close prices
aaplret=monthlyReturn(Ad(AAPL))
chart.Drawdown(aaplret,geometric=FALSE)
```

Drawdown Chart



Drawdown quoted

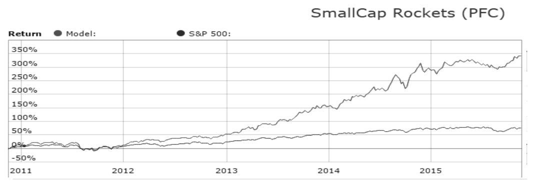
- From portfoliovisualizer.com



Portfolio	Initial Balance	Final Balance	CAGR	Std.Dev.	Best Year	Worst Year	Max. Drawdown
Timing Portfolio	$10,000	$12,812	3.79%	10.42%	23.69%	-9.40%	-13.73%
Buy & Hold Portfolio	$10,000	$17,839	9.07%	12.68%	23.69%	-9.95%	-20.58%

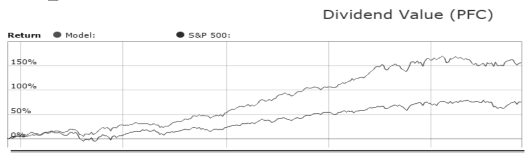
Example (portfoliocafe.com)

Members – please click link to see holdings



Total Return	15,095.29%	154.27%	215.16%	49.41%
Annualized Return	48.37%	7.60%	46.61%	14.32%
Max Drawdown	-29.72%	-56.78%	-17.08%	-12.35%
Standard Deviation	22.17%	14.09%	20.37%	10.61%
Sharpe Ratio	1.86	0.49	1.97	1.30
Sortino Ratio	2.81	0.63	2.98	1.77
Correlation with S&P 500	0.55	-	0.71	-
R-Squared	0.31	-	0.50	-
Beta	0.87	-	1.36	-

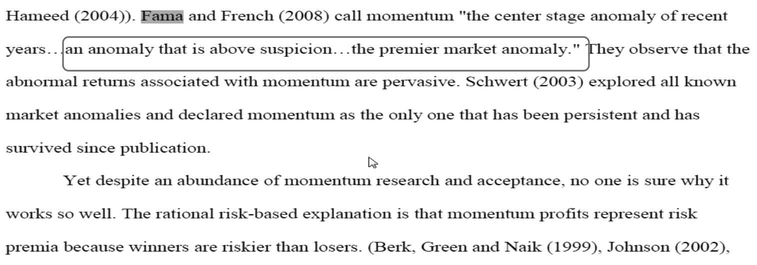
Example



Risk Measurements	Since Inception		Trailing 3 Year	
	Model	S&P 500	Model	S&P 500
Total Return	607.63%	44.55%	75.97%	49.41%
Annualized Return	26.67%	4.55%	20.73%	14.32%
Max Drawdown	-20.52%	-56.78%	-9.84%	-12.35%
Standard Deviation	13.09%	16.37%	10.46%	10.61%
Sharpe Ratio	1.89	0.32	1.95	1.30
Sortino Ratio	2.64	0.41	2.68	1.77
Correlation with S&P 500	0.63	-	0.60	-
R-Squared	0.39	-	0.36	-
Beta	0.50	-	0.59	-
Alpha (annualized)	24.60%	-	12.98%	-

One curious use for mean and std dev

- Momentum!
- Momentum is big



Hameed (2004)). Fama and French (2008) call momentum "the center stage anomaly of recent years…an anomaly that is above suspicion…the premier market anomaly." They observe that the abnormal returns associated with momentum are pervasive. Schwert (2003) explored all known market anomalies and declared momentum as the only one that has been persistent and has survived since publication.

Yet despite an abundance of momentum research and acceptance, no one is sure why it works so well. The rational risk-based explanation is that momentum profits represent risk premia because winners are riskier than losers. (Berk, Green and Naik (1999), Johnson (2002),

Lets go and play with http://www.etfreplay.com/backtest_rs.aspx

Density Estimation..Why be Normal?

- Density Estimation refers to determining a probability density for a given set of data.
- This can be as simple as assuming the data is normal, and using the sample mean and standard deviation as parameters.
- Or it can involve some fancy math.

BTW, History of Density Estimation

Kernel density estimation

From Wikipedia, the free encyclopedia

In statistics, **kernel density estimation** is a non-parametric way of estimating the probability density function of a random variable. Kernel density estimation is a fundamental data smoothing problem where inferences about the population are made, based on a finite data sample. In some fields such as signal processing and econometrics it is also known as the **Parzen-Rosenblatt window** method, after Emanuel Parzen and Murray Rosenblatt, who are usually credited with independently creating it in its current form,[1][2].

What do densities require?

- A probability density is a continuous function f(x) so that
- f(x) >=0 for all values of x
- The function f(x) integrates to 1.

The logspline package (unique to R)

- Library(logspline)
- Performs a semi-parametric density estimate, then lets you sample from the estimate.
- Pretty cool routine.
- What????

The Stone-Weierstrass Theorem

- From Wiki:

In mathematical analysis, the **Weierstrass approximation theorem** states that every continuous function defined on an interval [*a*,*b*] can be uniformly approximated as closely as desired by a polynomial function. Because polynomials are among the simplest functions, and because computers can directly evaluate polynomials, this theorem has both practical and theoretical relevance, especially in polynomial interpolation. The original version of this result was established by Karl Weierstrass in 1885.

Example

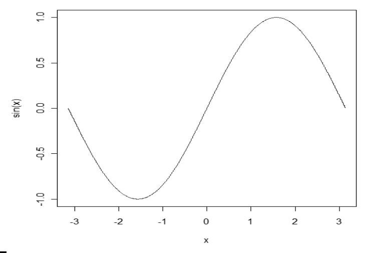
- Consider the function sin(x)
- Consider also the function P(x)

$$P(x) = x - \frac{1}{6}x^3 + \frac{1}{120}x^5 - \frac{1}{5040}x^7 + \frac{1}{362,880}x^9$$

- On the next slide is a graph of these two functions.

Can’t tell them apart

```
> plot(x,sin(x),col="red",type="l")
> lines(x,x-x^3/6+x^5/120-x^7/5040+x^9/362880,col="blue")
```



Approximate Densities

- We can do a similar approximation for any density.
- That is, given a set of data, we can find a polynomial approximation for the density of the underlying distribution.
- This is a powerful technique that we will use when we do Monte Carlo simulation.

The logspline package

- The package in R is called logspline
- It allows one to fit a density estimate then
  - Calculate Probabilties
  - Generate Random Observations
- Very powerful routine
  - logspline() – fit the density estimate
  - dlogspline() - density
  - rlogspline() – generate random values

Example

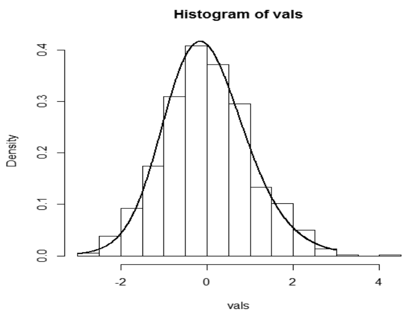
- Fit a density to some data

```
> vals=rnorm(100)
> fit=logspline(vals)
```

- Draw a histogram and overlay the estimated density.

```
> hist(vals,prob=TRUE)
> x=seq(-3,3,.01)
> lines(x,dlogspline(x,fit))
```

The Estimated Density



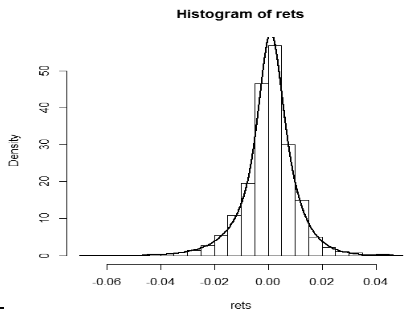
From a histogram of 100 values, it comes close to reproducing the underlying density

Model historical stock returns

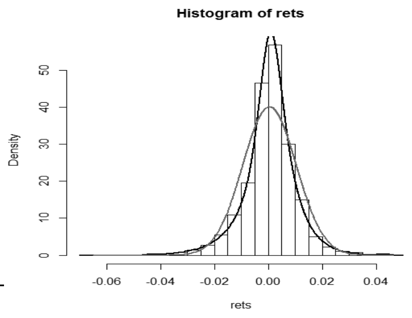
- Just a few lines of code

```
> getSymbols("SPY",from="2010-01-01")
[1] "SPY"
> rets=dailyReturn(Ad(SPY))
> rets=as.numeric(rets)
> fit=logspline(rets)
```

The resulting density



Compare with the Normal



Don't make assumptions like normality! Let your data do the talking.

Compare Probabilities

- What is the probability of a negative daily return?

```
> pnorm(0,mean(rets),sd(rets))
[1] 0.4794895

> plogspline(0,fit)
[1] 0.4441395
```

Can draw from this density

- A powerful ability of the logspline routine is that one **can draw random values** from the fitted density function.
- This is easily done using the rlogspline routine.
- Good way to simulate historical stock returns.
- More details when we cover Monte Carlo Simulation.

Playing with Density Estimation – Risk Measures

- We have seen that the standard deviation of returns is a useful risk measure.
- What if we calculate the probability of losing money and use that as a risk measure?

Example

- Compare AAPL, GOOG and JNJ
- By standard deviation

```
> sqrt(var(aaplret))
                 monthly.returns
monthly.returns        0.1120790
> sqrt(var(googret))
                 monthly.returns
monthly.returns        0.1016935
> sqrt(var(jnjret))
                 monthly.returns
monthly.returns       0.04568969
```

Compare by P(ret < 0)

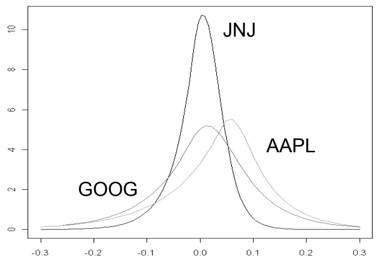
- Compute probability of a negative return

```
> fit.appl=logspline(as.numeric(aaplret))
> fit.goog=logspline(as.numeric(googret))
> fit.jnj=logspline(as.numeric(jnjret))

> plogspline(0,fit.appl)
[1] 0.3064950
> plogspline(0,fit.goog)
[1] 0.4355341
> plogspline(0,fit.jnj)
[1] 0.4694262
```

Riskiest??

Remember Fat Tails



This is why VaR is a useful measure-stay tuned for that

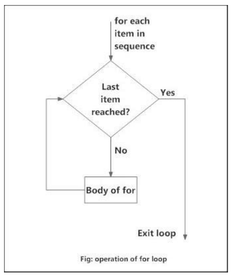
Aside: Looping in R

- We are about to start discussing two different yet similar ideas:
  - ❑ Resampling (the bootstrap)
  - ❑ Simulation
- For both of these ideas, we need to understand the basic idea of Looping.

Loops

- We loop when the same or similar task needs to be performed multiple times.
- Looping is also called iteration
- There are different loops such as `for loops` and `while loops`…we will concentrate on for loops.

Basic Loop Flowchart and Code



```
> for(i in 1:10) {
+    print(i*i)
+ }
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
[1] 49
[1] 64
[1] 81
[1] 100
```

For Loop

- Repeats a line or lines (known as a block) of code until:
  - (for loop) a count limit is reached

```
for (i in 1:10) {
  ... code ...
}
```

- the above loop runs 10 times
- '{' and '}' enclose code looped
- the variable i updated in the loop to values in the sequence 1:10
- i is called "the counter"; any letter can be used.

Can call a function inside a loop

- library(numbers)
- Has a function that checks if a number is prime [`isPrime()`]

```
> isPrime(123)
[1] FALSE
> isPrime(997)
[1] TRUE
```

Check the first 1000 numbers

```
> for(i in 1:1000) {
if (isPrime(i)) print(paste(i,"Is a prime number"))
}

[1] "2 Is a prime number"
[1] "3 Is a prime number"
[1] "5 Is a prime number"
[1] "7 Is a prime number"
[1] "11 Is a prime number"
[1] "13 Is a prime number"
[1] "17 Is a prime number"
[1] "19 Is a prime number"
```

The White Bull Example

- The following is a very simple R program that was posted on **ibankcoin.com** (a hoot of a site to read).
- All it does is check if a stock price is above both the 50 and 200 day moving average, and if the 50 day moving average is above the 200 day moving average.
- It is a good example to study for how to write a basic R program.

The Code

```
white_bull = function(ticker) {

    require("quantmod")

    x = getSymbols(ticker, auto.assign=FALSE)

    c             = x[,4]
    c50           = SMA(c, n=50)
    c200          = SMA(c, n=200)

    last_c        = tail(c, n=1)
    last_c50      = tail(c50, n=1)
    last_c200     = tail(c200, n=1)

    if (last_c > last_c50 && last_c > last_c200 && last_c50 > last_c200)
             { print(paste(ticker,"is in a bull market")) }
    else
       {  print(paste(ticker,"is NOT in a bull market")) }

}
```

Cut and Paste it into R

```
> white_bull("AAPL")
[1] "AAPL is in a bull market"
> white_bull("LCI")
[1] "LCI is in a bull market"
> white_bull("ANF")
[1] "ANF is NOT in a bull market"
> white_bull("IBM")
[1] "IBM is NOT in a bull market"
```

Loop over many Stocks

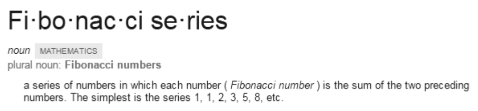
```
> mystocks=c("GOOG","PG","MMM","MO","ALL","AMZN","CSCO","C","FB")

> for(stock in mystocks) white_bull(stock)
[1] "GOOG is in a bull market"
[1] "PG is in a bull market"
[1] "MMM is NOT in a bull market"
[1] "MO is NOT in a bull market"
[1] "ALL is NOT in a bull market"
[1] "AMZN is in a bull market"
[1] "CSCO is in a bull market"
[1] "C is in a bull market"
[1] "FB is in a bull market"
```

Note you don't have to have a numerical loop; you can loop over any kind of list.

Storing Stuff

- We often want to store the output of what we are doing inside the loop
- Suppose we want to write a program to compute Fibonnaci Numbers



Fi·bo·nac·ci se·ries

noun MATHEMATICS
plural noun: **Fibonacci numbers**

a series of numbers in which each number (*Fibonacci number*) is the sum of the two preceding numbers. The simplest is the series 1, 1, 2, 3, 5, 8, etc.

R Code

```
len <- 10
fibvals <- numeric(len)
fibvals[1] <- 1
fibvals[2] <- 1
for (i in 3:len) {
fibvals[i] <- fibvals[i-1]+fibvals[i-2]
}
```

Running the Code

```
> len <- 10
> fibvals <- numeric(len)
> fibvals[1] <- 1
> fibvals[2] <- 1
> for (i in 3:len) {
+    fibvals[i] <- fibvals[i-1]+fibvals[i-2]
+ }
> fibvals
 [1]  1  1  2  3  5  8 13 21 34 55
```

Make it into a function

```
myfun=function(len) {

fibvals <- numeric(len)
fibvals[1] <- 1
fibvals[2] <- 1
for (i in 3:len) {
   fibvals[i] <- fibvals[i-1]+fibvals[i-2]
}
return(fibvals)
}
```

Function Output

```
> myfun(10)
 [1]  1  1  2  3  5  8 13 21 34 55
> myfun(20)
 [1]    1    1    2    3    5    8   13   21   34   55   89  144  233  377  610
[16]  987 1597 2584 4181 6765
```

MA Routine

- Buy if above 200day MA, sell otherwise.
- Compare to buy and hold.
- Note the 200day MA needs 200 days of data to compute, so we start our system on day 201.
- Note also there are 250 days in a trading year.

The Code

- Initialization

```
startdate="2010-01-01"
enddate = "2016-09-01"
malength=200
ticker="AAPL"

stockdata=getSymbols(ticker,from=startdate,to=enddate,auto.assign=FALSE)
sp=as.numeric(Ad(stockdata))
ndays=length(sp)
ma=SMA(sp,malength)

signal="inCash"
buyprice=0
sellprice=0
mawealth=1
```

The Loop (EOD transactions)

```
for(d in (malength+1):ndays) {
if((sp[d]>ma[d]) && (signal=="inCash")) {
   buyprice=sp[d]
   signal = "inStock"
   ##print(paste("Buy Price = ",buyprice))
}

if(((sp[d]<ma[d]) || (d==ndays)) && (signal=="inStock")) {
   sellprice=sp[d]
   signal = "inCash"
   mawealth=mawealth*(sellprice/buyprice)
   ###print(paste("Sell Price = ",sellprice))
}
}

bhwealth=sp[ndays]/sp[malength+1]
```

Finally calculate CAGR

- Need to determine number of years and the calculate CAGR from total return

```
nmyears=(ndays-malength)/250
print(paste("MA CAGR = ", (1+mawealth)^(1/nmyears)-1))
print(paste("BH CAGR = ", (1+bhwealth)^(1/nmyears)-1))
```

Running it (made it into a function)

```
 my200ma("AAPL")
[1] "MA CAGR =  0.264198809565228"
[1] "BH CAGR =  0.243881413781211"

> my200ma("ANF")
[1] "MA CAGR =  0.100971327101597"
[1] "BH CAGR =  0.0660409074273329"

> my200ma("PG")
[1] "MA CAGR =  0.139747529804917"
[1] "BH CAGR =  0.18225526748927"
```

Moving On

- Loops are useful
- There are fancier ways to do things in R other than using loops [R geeks will says “loops are expensive”]
- But for now, we will use loops because they are easy to understand.

Confidence Interval for correlation

- Last hw-CI for correlation…….no one remembers the formula.
- In fact, we actually need the assumption of normality to compute the CI….
- It would be nice if there was a more general way to produce confidence intervals without having to assume normality (or formulas).
- We now will learn how to create confidence intervals by the method of resampling.

More Concisely…

- Point estimate says:
  “what do you think?”
- Variability of the point estimate says: “how sure are you?”
- Traditional approaches

  Use distributional assumptions to construct confidence intervals
- Is there an easier – and more flexible – way?

Enter the Bootstrap

- In the late 70’s the statistician Brad Efron made an ingenious suggestion.
- Most (sometimes all) of what we know about the “true” probability distribution comes from the data.
- So let’s treat the data as a *proxy* for the true distribution.
- We draw multiple samples from this proxy…
  - This is called “resampling”.
- And compute the statistic of interest on each of the resulting pseudo-datasets.

Brad Efron

- Brad Efron from Stanford invented the bootstrap (and proved this simple yet elegant idea works).

Efron Dad Mom Me

Philosophy

- "[Bootstrapping has] requires very little in the way of modeling, assumptions, or analysis, and can be applied in an automatic way to any situation, no matter how complicated".
- "An important theme is the substitution of raw computing power for theoretical analysis"

  --Efron and Gong 1983
- Bootstrapping fits very nicely into the "data mining" paradigm.

Resampling

- In resampling, the object of interest is a statistic which is a function of the data Θ(x1,x2,x3,…..).
- This might be
  - ❑ Θ(x1,x2,x3,…..) = mean(x's)
  - ❑ Θ(x1,x2,x3,…..) = median(x's)
  - ❑ Θ(x1,x2,x3,y1,y2,y3…..) = correlation(x,y)

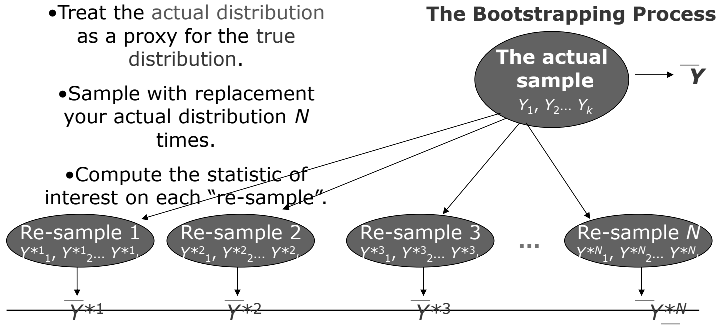
Or VaR, Omega, Sharpe, etc….

How resampling works

- One literally resamples the data, that is, draws random samples of size *n* from the observed data set.
- For each resampled data set, one can compute Θ_i(x1,x2,x3,…..)
- This is done say 10000 times, producing 10000 Θ_i's.
- A confidence interval is then calculated.

The Basic Idea

- Treat the actual distribution as a proxy for the true distribution.
- Sample with replacement your actual distribution *N* times.
- Compute the statistic of interest on each "re-sample".



- {Y^*} constitutes an estimate of the *distribution* of $\overline{Y}$.

Summary

- The empirical distribution – your data – serves as a proxy to the "true" distribution.
- "Resampling" means (repeatedly) sampling with replacement.
- Resampling the data is analogous to the process of drawing the data from the "true distribution".
- We can resample multiple times
  - Compute the statistic of interest *T* on each re-sample
  - We get an estimate of the distribution of *T*.

How the CI is calculated

- There are several ways to calculate a confidence interval once 10000 estimates are obtained:
- Empirical CI; sort the values and take the middle 95%
- Normality; mean +/- 1.96*(std dev)
- The middle 95% from a density estimate.

Example with the mean

- The usual 95% confidence interval for a population mean from a sample of size *n* is given by

$$\bar{x} \pm t_{(.025,n-1)} \frac{s}{\sqrt{n}}$$

Example: the sample mean

- Suppose we want to create a confidence interval for mean exam score from the following data:

```
> mydata=c(78,88,62,90,99,86,85,92,94,75,53,92)
> t.test(mydata)

        One Sample t-test

data:  mydata
t = 20.99, df = 11, p-value = 0.000000000318
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 74.14760 91.51907
sample estimates:
mean of x
 82.83333
```

Quick aside-the sample function in R

```
      > data
[1]  4 10  6  2 15
```

```
> sample(data,5,replace=T)
   [1] 10  2 15  4  4
> sample(data,5,replace=T)
   [1]  2 10  2 10  2
> sample(data,5,replace=T)
   [1]  2  2 15  6  6
> sample(data,5,replace=T)
   [1]  4 15  2  2  2
> sample(data,5,replace=T)
   [1]  6  6  4 15  6
> sample(data,5,replace=T)
   [1] 10  4  4 15  2
> sample(data,5,replace=T)
   [1]  2  4 15  4 15
> sample(data,5,replace=T)
   [1] 10  4 15 15  2
> sample(data,5,replace=T)
   [1]  2  6  6 15 10
```

If you didn't know the formula…

- Resample the data a large number of times,
- Computing the mean return for each resample.

```
n=10000
vals=1:n
exam=c(78,88,62,90,99,86,85,92,94,75,53,92)
sampsize=length(exam)
for(i in 1:n) {
 resampvals=sample(exam,sampsize,replace=TRUE)
 vals[i]=mean(resampvals)
}
```

Empirical 95% Confidence Interval

- We take the 10000 computed means and sort them from smallest to largest
- The empirical 95% confidence will be the values between the lower 2.5 percentile and upper 97.5%

```
> ord=order(vals)
> svals=vals[ord]
> svals[250]
[1] 74.75
> svals[9750]
[1] 89.66667
```

```
95 percent confidence interval:
 74.14760 91.51907
```

Close!

CI based on the normal distribution

- The mean of the 10000 resampled values +/- 1.96 * standard deviation

```
> mean(vals)-1.96*sqrt(var(vals))
[1] 75.44916
> mean(vals)+1.96*sqrt(var(vals))
[1] 90.2535
```

```
95 percent confidence interval:
 74.14760 91.51907
```

CI based on a density estimate

- Fit a density estimate to the resampled values, then find the inner 95% of that resulting density.

```
> fit=logspline(vals)
> qlogspline(.025,fit)
[1] 74.96168
> qlogspline(.975,fit)
[1] 89.67376
```

```
95 percent confidence interval:
 74.14760 91.51907
```

Your statistical toolbox

- Two important techniques in your toolbox
- Resampling
  - ❑ Use it whenever you devise a new function of your data (like Sharpe ratio) and want to understand its variability
- Density estimation
  - ❑ Use it to model historical returns, output of a process, etc.…

Resampling = the Bootstrap

- The Bootstrap is another name for resampling "pick yourself up by your bootstraps…"
- Let the data tell you about itself.

"Pulling oneself up by one's bootstraps"

"I found myself stunned, and in a hole nine fathoms under the grass, when I recovered, hardly knowing how to get out again. Looking down, I observed that I had on a pair of boots with exceptionally sturdy straps. Grasping them firmly, I pulled with all my might. Soon I had hoist myself to the top and stepped out on terra firma without further ado." -- Campaigns and Adventures of Baron Munchausen, 1786.

More Interesting Examples

- We've seen that bootstrapping (resampling) replicates a result we know to be true from theory.
- Often in the real world we either don't know the 'true' distributional properties of a random variable…
- …or are too busy to find out.
- This is when bootstrapping really comes in handy.

The Bootstrap in R

- Although resampling/bootstrap code is easy to write, there is a bootstrap routine built into R.
- library(boot)
- The function call:

```
bootstrap(x,f,R=1000)
```

The data

The function to bootstrap

The number of bootstraps

The function to bootstrap

- Need to write a function that takes as arguments d (the data) and i (which rows to grab).
- For bootstrapping the mean, the function would be

```
f=function(d, i)
     {return(mean(d[i]))}
```

The Mean Example Again

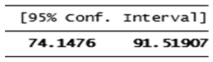
- Running this in R

```
> bfit=boot(exam,f,R=1000)
> boot.ci(bfit)
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 1000 bootstrap replicates

CALL :
boot.ci(boot.out = bfit)

Intervals :
Level      Normal              Basic
95%   (75.45, 89.98 )   (76.33, 90.66 )

Level     Percentile            BCa
95%   (75.00, 89.33 )   (73.27, 88.58 )
```



```
[95% Conf. Interval]
  74.1476    91.51907
```

Bootstrap the Correlation

- Start with monthly data

```
getSymbols("MRK",from="2012-01-01")
getSymbols("LULU",from="2012-01-01")
mrk.ret=as.numeric(monthlyReturn(Ad(MRK)))
lulu.ret=as.numeric(monthlyReturn(Ad(LULU)))
 > library(nortest)
 > ad.test(mrk.ret)
         Anderson-Darling normality test
 data:  mrk.ret
 A = 0.18428, p-value = 0.9034

 > ad.test(lulu.ret)
         Anderson-Darling normality test
 data:  lulu.ret
 A = 0.23419, p-value = 0.7819
```

The Usual Interval

- The data appears normally distributed so we will calculate the usual interval

```
> cor.test(mrk.ret,lulu.ret)

        Pearson's product-moment correlation

data:  mrk.ret and lulu.ret
t = -0.82268, df = 43, p-value = 0.4152
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.4032802  0.1754632
sample estimates:
       cor
-0.1244821
```

Bootstrap

- Have to write function

```
f <- function(d, i){
        d2 <- d[i,]
        return(cor(d2[,1], d2[,2]))
}
mydata=cbind(mrk.ret,lulu.ret)
bfit=boot(mydata,f,R=1000)
boot.ci(bfit)
```

Similar!

```
Intervals :
Level      Normal              Basic
95%   (-0.4243,  0.1680 )   (-0.4428,  0.1643 )

Level     Percentile            BCa
95%   (-0.4132,  0.1938 )   (-0.4283,  0.1579 )
```

Now Daily Data

- Daily data is not normal

```
> getSymbols("MRK",from="2012-01-01")
[1] "MRK"
> getSymbols("LULU",from="2012-01-01")
[1] "LULU"
> mrk.ret=as.numeric(dailyReturn(Ad(MRK)))
> lulu.ret=as.numeric(dailyReturn(Ad(LULU)))
> ad.test(mrk.ret)
        Anderson-Darling normality test
data:  mrk.ret
A = 6.5887, p-value = 3.664e-16

> ad.test(lulu.ret)
        Anderson-Darling normality test
data:  lulu.ret
A = 15.78, p-value < 2.2e-16
```

Correlation Confidence Interval

- Takes longer since much more data

```
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 1000 bootstrap replicates

CALL :
boot.ci(boot.out = bfit)

Intervals :
Level      Normal              Basic
95%   ( 0.0067,  0.2160 )   ( 0.0176,  0.2275 )

Level     Percentile            BCa
95%   (-0.0027,  0.2071 )   (-0.0410,  0.1903 )
Calculations and Intervals on Original Scale
Some BCa intervals may be unstable
```

The point is you can't use the regular interval since not normal data

Remember the Bootstrap

- The bootstrap (resampling) is an incredibly powerful tool to keep around.
- It is easy to explain to people
- Enables one to create confidence intervals for anything you can estimate from the data, (correlation, VaR, Omega, whatever) without worrying about complex math.

Not all statistics behave well when bootstrapped

- Many Ω's "work well" (give good CIs and inferences) in bootstraps: means, variances, typical inferential stats, eigenvalues, etc.
- Some Ω's require larger n's to work well: medians, percentiles
- Some Ω's don't work well at all: min, max, unique values
- How to know which ones will work? The higher the variance and skewness of Ω, the less well bootstraps work. You might try simulating to get an idea about these.

Monte Carlo- Historical Note

- The name Monte Carlo simulation comes from the computer simulations performed during the 1930s and 1940s to estimate the probability that the chain reaction needed for an atom bomb to detonate would work successfully.
- The physicists involved in this work were big fans of gambling, so they gave the simulations the code name Monte Carlo.

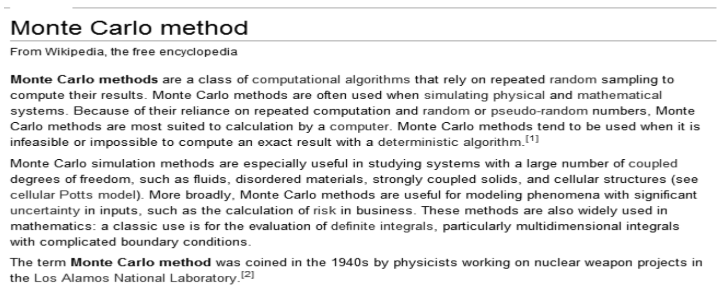
Uses of Monte Carlo Simulation

- Ford, Proctor and Gamble, Pfizer, Bristol-Myers Squibb, and Eli Lilly use simulation to estimate both the average return and the risk factor of new products.
- Proctor and Gamble uses simulation to model and optimally hedge foreign exchange risk.
- Sears uses simulation to determine how many units of each product line should be ordered from suppliers—for example, the number of pairs of Dockers trousers that should be ordered this year.
- Financial planners use Monte Carlo simulation to determine optimal investment strategies for their clients' retirement.

The Monte Carlo Method

- ❑ The method is usually used to understand how a process or estimator will perform "in practice".
- ❑ The Monte Carlo method is often performed when the computational resources of a researcher are more abundant than the researchers mental resources.

Wikipedia



Monte Carlo method

From Wikipedia, the free encyclopedia

Monte Carlo methods are a class of computational algorithms that rely on repeated random sampling to compute their results. Monte Carlo methods are often used when simulating physical and mathematical systems. Because of their reliance on repeated computation and random or pseudo-random numbers, Monte Carlo methods are most suited to calculation by a computer. Monte Carlo methods tend to be used when it is infeasible or impossible to compute an exact result with a deterministic algorithm.[1]

Monte Carlo simulation methods are especially useful in studying systems with a large number of coupled degrees of freedom, such as fluids, disordered materials, strongly coupled solids, and cellular structures (see cellular Potts model). More broadly, Monte Carlo methods are useful for modeling phenomena with significant uncertainty in inputs, such as the calculation of risk in business. These methods are also widely used in mathematics: a classic use is for the evaluation of definite integrals, particularly multidimensional integrals with complicated boundary conditions.

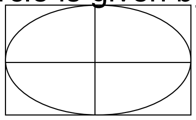
The term **Monte Carlo method** was coined in the 1940s by physicists working on nuclear weapon projects in the Los Alamos National Laboratory.[2]

Example: Estimating π

- Did you know that you can calculate π (3.14159..) by throwing darts at a dartboard ?
- Unfortunately, you have to be pretty bad at darts for it to work, though.

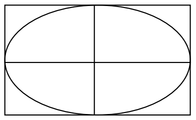
Geometry Review

- Suppose we have a circle of radius 1, with a center at (0,0).
- Then the unit circle is given by the equation $x^2+y^2=1$.

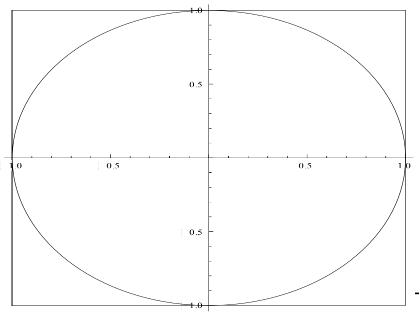


The Set Up

- Suppose our dartboard is a 2x2 square and we inscribe a circle of radius 1 inside the square, with a center at (0,0).



Computing π: Unit circle inscribed in unit square



- The square has area 4
- Unit circle has area π

The Algorithm

- We randomly throw darts at this square and keep track of how many land inside the circle. That would give us a relative estimation of what percentage of the box is in the circle.
- The area of the box is 4 and the area of the circle is π.
- So the ratio of circle area to box area is π/4.
- Hence, the percentage of darts that land inside the circle should be approximately π/4.

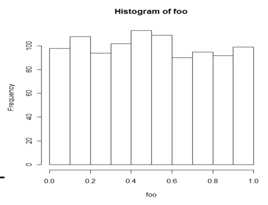
The runif() command

- To simulate throwing darts at the dartboard, we need to learn about the R command runif():

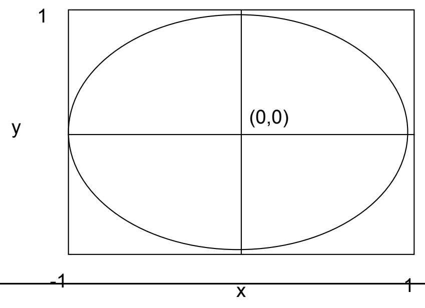
R Command: runif(n)
uniform random number generator that returns
n random numbers between 0 and 1.

Examples

```
> runif(5)
[1] 0.7000905 0.6717379 0.9857697 0.6081811 0.8611683
>
> runif(5)
[1] 0.1719788 0.5086680 0.6567634 0.1922734 0.8315596
>
> foo=runif(1000)
> hist(foo)
>
```



Our circle is as follows:



How do we simulate darts thrown at this board ?

The location of the dart is given by its (x,y) point.

Ponder

- What do the following R commands
- produce?

- `x = 2*runif(1)-1`
- `y = 2*runif(1)-1`

- Lets plot this and see

Some R Code

- Consider the following code

```
n=500
x=matrix(nrow=n,ncol=1)
y=matrix(nrow=n,ncol=1)

for(i in 1:n) {
  x[i] = 2*runif(1)-1
  y[i] = 2*runif(1)-1
}
```

Let’s run it

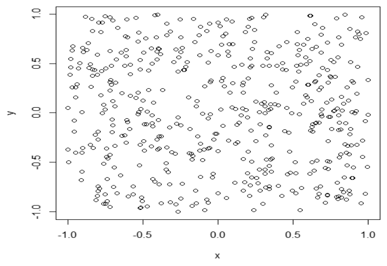
- I cut and paste it into R and get:

```
> n=500
> x=matrix(nrow=n,ncol=1)
> y=matrix(nrow=n,ncol=1)
>
> for(i in 1:n) {
+   x[i] = 2*runif(1)-1
+   y[i] = 2*runif(1)-1
+ }
>
```

That is, I get nothing….or do I????
New variables x and y are created

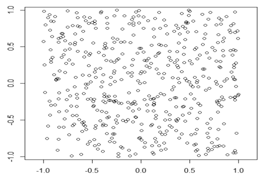
Plot the new variables

- plot(x,y)



Different plot command

- eqscplot(cbind(x,y))
- Makes the x and y axis the exact same length (1 inch in x direction=1 inch in y direction)

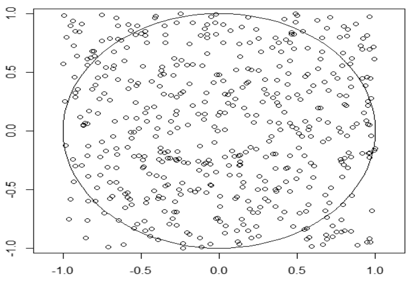


Add a circle

- This is a trick. We need a new package called “plotrix” and can then use its draw.circle routine.

- `library(plotrix)`
- `draw.circle(0,0,1)`

The Graph



So far the code looks to be working

Code Fragment

- Examine the following code fragment

```
x = 2*runif(1)-1
y = 2*runif(1)-1
if (x*x+y*y<=1) numincircle = numincircle+1
```

- What does this code do ?

The Complete Function

```
simulpi = function(n) {

numincircle = 0

for(i in 1:n) {
  x = 2*runif(1)-1
  y = 2*runif(1)-1
  if (x*x+y*y<1) numincircle = numincircle+1
}
cat("Number of simulations = ",n,"\n")
cat("Estimate of PI = ",4*numincircle/n,"\n")
}
```

Keep count of darts inside circle

do n simulations

Output

```
> simulpi(10)
Number of simulations =  10
Estimate of PI =  3.2
> simulpi(100)
Number of simulations =  100
Estimate of PI =  3.04
> simulpi(1000)
Number of simulations =  1000
Estimate of PI =  3.136
> simulpi(1000)
Number of simulations =  1000
Estimate of PI =  3.184
> simulpi(100000)
Number of simulations =  1e+05
Estimate of PI =  3.1472
```

1000 iterations-why aren't the values the same each time ?