# **Monte Carlo Methods.**

Probabilistic models are often too complex to be approached analytically, but they can be analyzed numerically. The key to these numerical methods is their capacity to sample from *distributions* in order to identify for each parameter the values that allow fitting a specific model to the experimental data. In most cases, the sampling of complex distributions involves sampling much simpler distributions as a step.

The MATLAB Statistics Toolbox supports many basic distributions, for which is possible to calculate the *probability density function* (*pdf*), the *cumulative probability density function* (*cdf*) (CHAPTER 20, SPECIAL TOPICS), and from which is possible to *draw random samples*. For example, the Normal *pdf* is defined with respect to its *mean* and *standard deviation*:

We can visualize the *normal distribution X~N*(μ,σ) (read as ‘X distributed as *N*(μ,σ)) with *mean,* μ=60, and *standard deviation*, σ=20.

Mean and standard deviation:

mu = 60;

sigma = 20;

Minimum and maximum x values for *pdf* and *cdf* plot, number of points and number of draws:

xmin\_norm = -20; xmax\_norm = 140;

npoints = 100; ndraws = 10000;

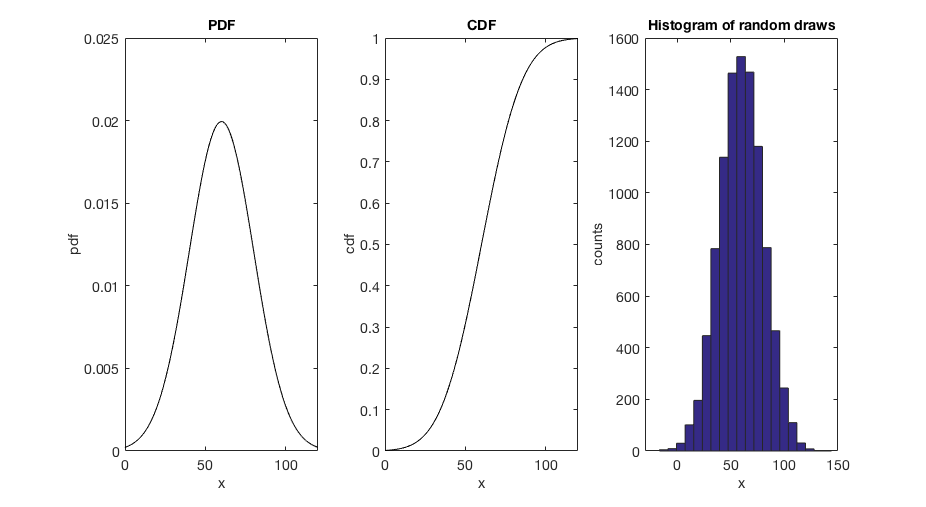
Here we create a set of values ranging from xmin to xmax, and we calculate the pdf and cdf:

x\_norm = linspace(xmin\_norm,xmax\_norm,npoints);

pdf\_norm = normpdf(x\_norm,mu,sigma);

cdf\_norm = normcdf(x\_norm,mu,sigma);

Here we draw k random numbers from the distribution:

y\_norm = normrnd(mu,sigma,ndraws,1);

Normal\_distribution = figure; clf;

subplot(1,3,1);

plot(x\_norm,pdf\_norm,'-k');

xlabel('x'); ylabel('pdf');

xlim([0 120]);ylim([0 0.025])

title('PDF');

subplot(1,3,2);

plot(x\_norm,cdf\_norm,'-k');

xlabel('x'); ylabel('cdf');

xlim([0 120]);

title('CDF');

subplot(1,3,3);

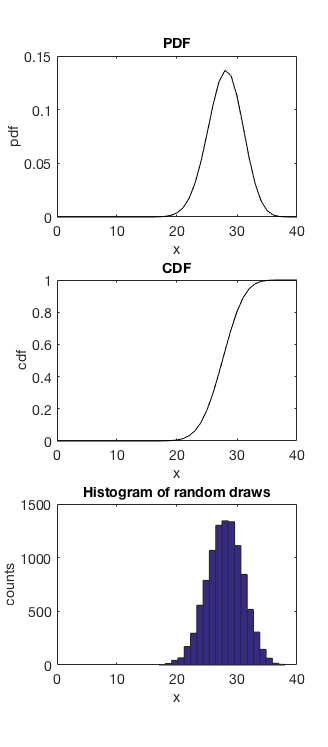
hist(y\_norm,20);

xlim([-30 150])

xlabel('x'); ylabel('counts');

title('Histogram of random draws');

Likewise we could analyze the *Binomial distribution, X~B*(n,p)*,* which is the distribution ofthe number of successes in *n* independent trials, when each trial has a probability of success *p*.

clc, close all

n = 40;

p = 0.7;

xmin\_bino = 0;

xmax\_bino = 40;

ndraws = 10000;

x\_bino = xmin\_bino:xmax\_bino;

pdf\_bino = binopdf(x\_bino,n,p);

cdf\_bino = binocdf(x\_bino,n,p);

y\_bino = binornd(n,p,ndraws,1);

Binomial\_distribution = figure; clf;

subplot(3,1,1);

plot(x\_bino,pdf\_bino,'-k');

xlabel('x'); ylabel('pdf');

xlim([0 xmax\_bino]);

title('PDF');

subplot(3,1,2);

plot(x\_bino,cdf\_bino,'-k');

xlabel('x'); ylabel('cdf');

xlim([0 xmax\_bino]);

title('CDF');

subplot(3,1,3);

hist(y\_bino,20);

xlim([0 xmax\_bino])

xlabel('x'); ylabel('counts');

title('Histogram of random draws');

**The inverse transform method.**

In the analysis of both distributions we have used an existing MATLAB function to draw samples from the distribution. However, in the more general case we may have the *pdf* and/or *cdf*, but we do not have a function to draw random samples from the distribution. The problem can be overcome by using the *inverse transform sampling*, a method to generate random numbers from *any* distribution given the *inverse* of its *cdf*. If only the *pdf* is available the *cdf* can be easily calculated with a summation (for discrete distributions), or integrating (for continuous distributions):

CDF from summation (discrete variable)

cdf\_bino\_alt = cumsum(pdf\_bino);

CDF from integration

dx = mean(diff(x\_norm))

for i = 1:length(pdf\_norm)

cdf\_norm\_alt(i) = dx\*trapz(pdf\_norm(1:i));

end

PDF from differentiation

pdf\_norm\_alt = gradient(cdf\_norm,dx)

As an example, we can apply the *inverse method* to the binomial distribution we just analyzed. To understand how the method works it is convenient to draw the *cdf* as a staircase:

CDF\_staircase = figure; clf;

stairs(x\_bino,cdf\_bino,'b');

xlim([0 xmax\_bino]);

ylim([0 1]);

ylabel('cdf');

xlabel('x');

grid on; box on

The idea is to sample *uniform random deviates u* (i.e., random numbers between 0 and 1: this sampling is offered in most programming languages) and to compare each random number against the *cdf*. The first element of the *cdf* larger or equal to the random number corresponds to the sampled outcome.

For just 1 draw we get:

hold on;

Uniform random deviate from 0 to 1

u = rand;

Index of the first cdf value >= u

xu\_ind = find( cdf\_bino >= u , 1 , 'first' );

Random sample

y\_bino = x\_bino(xu\_ind);

plot([0 y\_bino],[u u],'--k');

plot([y\_bino y\_bino],[u 0],'--k');

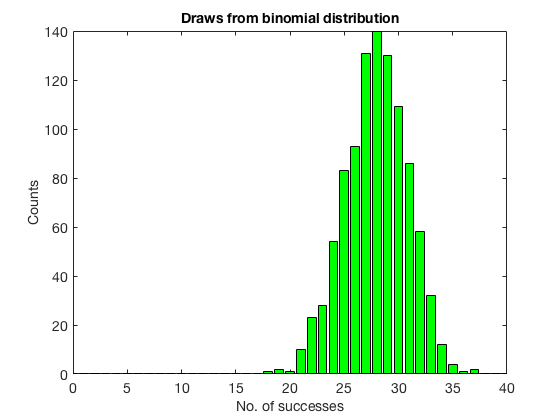
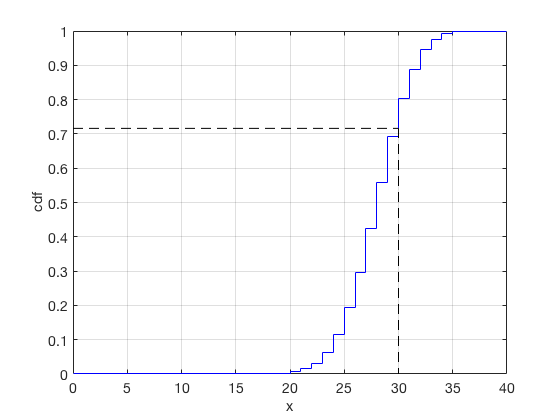
u = 0.7161

y = 30

We can easily draw thousands to obtain a nice histogram using a MATLAB function:

ndraws = 1000;

y\_bino = randsample(x,ndraws,true,pdf);



or using a *for* loop with the *inverse transform method*:

y\_bino = zeros(ndraws,1);

for i = 1:ndraws

u = rand;

xu\_ind = find( cdf\_bino > u , 1 , 'first' );

y\_bino(i) = x\_bino(xu\_ind);

end

Binomial\_dist\_hist = figure; clf;

counts = hist(y\_bino,0:xmax\_bino);

bar(x\_bino,counts,'g');

xlim( [0 40] );

xlabel('No. of successes' );

ylabel( 'Counts' );

title( 'Draws from binomial distribution' );

The *inverse transform method* can also be applied to distributions that are *continuous* instead of *discrete* like the binomial distribution. Consider as an example the *exponential distribution, X~exp(μ)*: this is the probability distribution of the times between events in a *Poisson process*, i.e. a process in which events occur continuously and independently at a constant average rate. In MATLAB this distribution *pdf* is defined as:

with *cdf*:

where *μ* is the mean of the distribution. In this case application of the *inverse transform methods* becomes self evident because we are using the actual inverse of the *cdf* (often referred to as the *quantile* distribution):

This leads to the following procedure to sample random numbers from an exponential distribution:

1. draw *u~U(0,1)*

2. set

3. iterate

For *μ=5* we get:

mu = 5;

Xmin and xmax only for plotting:

xmin = 0; xmax = 20;

npoints = 100;

ndraws = 10000;

x = linspace(xmin,xmax,npoints);

pdf\_exp = exppdf(x,mu);

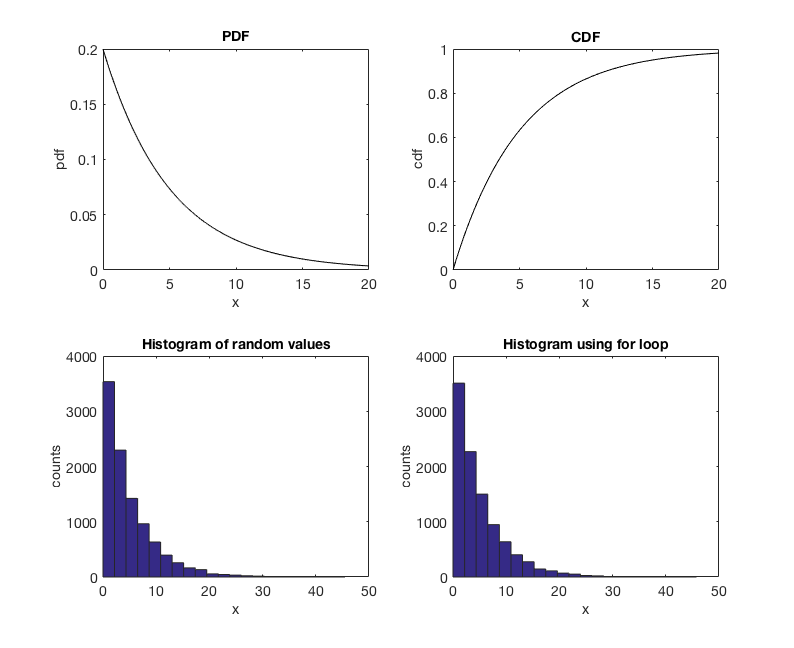
cdf\_exp = expcdf(x,mu);

Exponential\_dist = figure; clf;

subplot(2,2,1);

plot(x,pdf\_exp,'-k');

xlabel('x'); ylabel('pdf');

title('PDF');

subplot(2,2,2);

plot(x,cdf\_exp,'-k');

xlabel('x'); ylabel('cdf');

title('CDF');

Draw random numbers from the distribution using MATLAB function *exprnd*:

y = exprnd(mu,ndraws,1);

subplot(2,2,3);

hist(y,21);

xlabel('x'); ylabel('frequency');

title('Histogram of random values');

Alternatively, using a for loop:

X = zeros(10000,1);

U = rand(10000,1);

for i = 1:10000

X(i) = -log(1-U(i))\*mu;

end

subplot(2,2,4);

hist(X,21);

xlabel('x'); ylabel('counts');

title( 'Histogram using for loop' );

**Markov Chain Monte Carlo (MCMC).**

Until now we have looked at distributions of just one variable (univariate distributions). However, the application of probabilistic models to experimental data often requires the sampling of complex, high dimensional distributions (multivariate distributions). *Markov Chain Monte Carlo (MCMC)*, is a general computational approach that replaces analytic integration by summation over samples generated from iterative algorithms. We start by discussing the two main ideas underlying MCMC: *Monte Carlo integration*, and *Markov chains*.

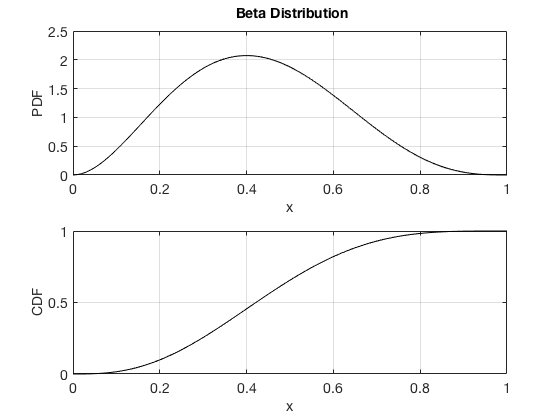
*Monte Carlo integration.* A frequent problem in probabilistic inference is to calculate the *expectation* *E*[*g(x)*]of a function *g(x)* of a random variable *x* (i.e., a univariate random variable). We are familiar with this operation: for example, if *g(x)* = *x*, we are calculating the mean of the distribution. In general, if *x* is continuous, the expectation is defined as:

if *x* is discrete:

Monte Carlo integration uses random *samples* to approximate the expectation of a distribution. Specifically, if we obtain a sufficiently large set of samples *xi*, *i* = *1*,…,*n*, drawn independently from the distribution *p*(*x*), we can approximate the expectations by a finite sum. The accuracy of the approximation can be made as accurate as desired by increasing *n*:

As an example, consider the *beta distribution X~Beta(a,b)*: this is a continuous distribution defined on the interval [0,1] by two positive parameters (*a* and *b*) that appear as exponents of the random variable and control the shape of the distribution:

where the *beta function*, , is a normalization constant to ensure that the total probability integrates to 1.



a = 3; b = 1;

xmin = 0;xmax = 1;

n = 100;k = 10000;

x = linspace(xmin,xmax,n);

pdf\_beta = betapdf(x,a,b);

cdf\_beta = betacdf(x,a,b);

figure;

subplot( 2,1,1 );h1 = plot( x , pdf\_beta , 'k-' );

xlabel( 'x' ); ylabel( 'PDF' );grid on

subplot( 2,1,2 );h2 = plot( x , cdf\_beta , 'k-' );

xlabel( 'x' ); ylabel( 'CDF' );grid on

[ax,h3] = suplabel('Beta Distribution','t',[.08 .08 .88 .88])

The *beta* distribution (CHAPTER 20, Special Topics) is often used in *Bayesian analysis* to describe initial knowledge (*prior probability)* concerning the (*posterior*) probability of success of a given event.

We can calculate the *mean* and *variance* of *X~Beta(3,4)* using Monte Carlo sampling and integration. To this end we draw random numbers from the *beta distribution*. An alternative to the *inverse method* is the *rejection method*:

1. choose a density *q* that is easy to sample from the interval [xmin xmax]. In this case we choose a uniform density, but it could be any density that can be made larger than the density we want to sample.

q = unifpdf(x,xmin,xmax);

2. find the smallest constant *c* > largest value of the pdf of the beta distribution.

c = max(pdf\_beta) + eps;

3. sample a uniform deviate *theta* from the interval [xmin xmax].

nsamples = 30000;

theta = unifrnd(xmin,xmax,nsamples,1);

4. sample a uniform deviate *u* from the interval [0 c].

u = c\*rand(nsamples,1);

5. accept theta if *u*<*p\_theta*

p\_theta = betapdf(theta,a,b);

theta\_accept = theta(u<p\_theta);

hist(theta\_accept,20);

xlabel( 'x' ); ylabel( 'frequency' );

title( 'Histogram using rejection method' );

The same result is obtained with just a few lines of MATLAB code:

betadistr = betarnd(a,b,k,1);

The mean and variance are:

a = 3; b = 4;

mean\_beta = mean(betadistr);

var\_beta = var(betadistr);

which correspond quite well to the theoretical values:

mean\_beta = a/(a+b);

var\_beta = (a\*b)/((a+b)^2 \* (a+b+1));

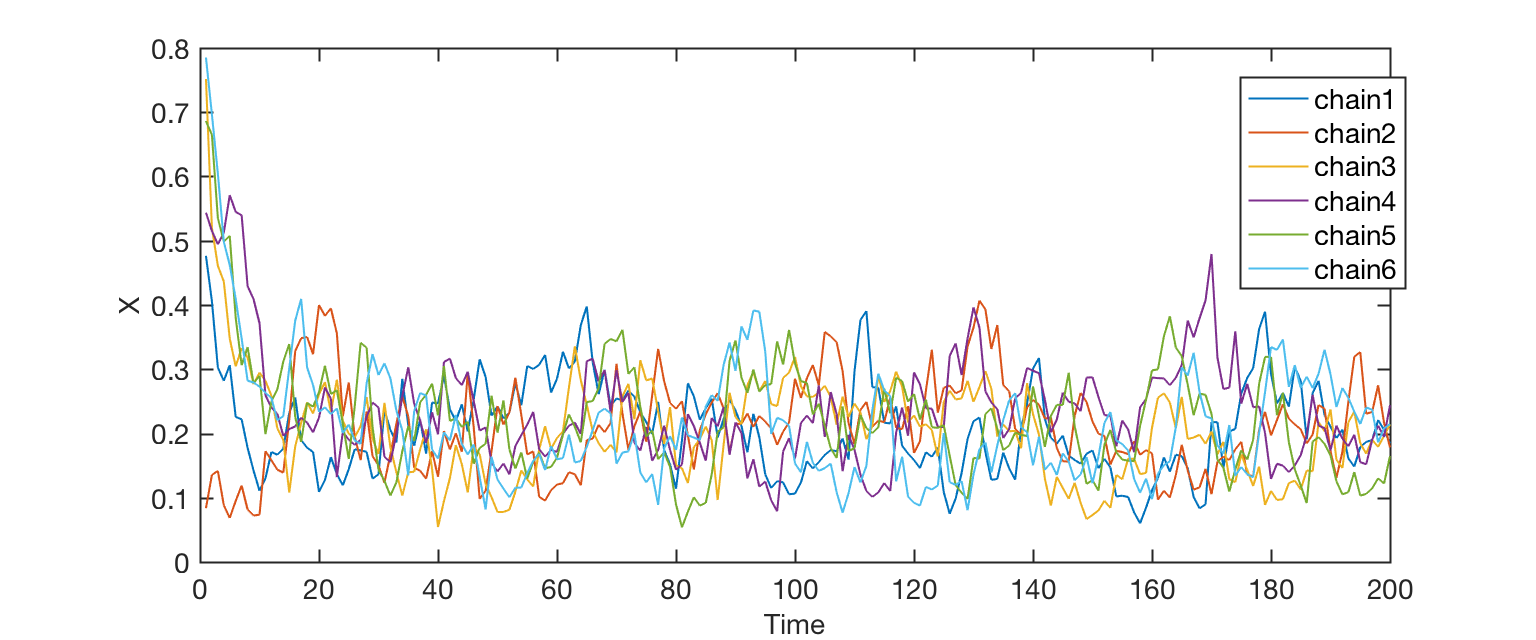
Markov Chains. A Markov chain is a stochastic process in which a *transition function*:

*f* = *p*(*x(t)*|*x(t−1)*)

determines the transition from a state *x(t-1)* to the next state, *x(t)*. Iterative application of the function creates a sequence of states:

called a *Markov chain* or simply a *chain*. For example, the *transition function* could be a *beta pdf:*

*200\*(0.8\*+ 0.04), b =* *100\*(1-0.8\* - 0.04)*

and we can implement the cycle for multiple random initial values of *u* at the same time*.*

nchains = 6;

ndraws = 200;

X = zeros(ndraws,nchains);

X(1,:) = rand(1,nchains);

for i = 1:ndraws-1

a = 200\*(0.8\*X(i,:) + 0.04);

b = 200\*(1-0.8\*X(i,:) -0.04);

X(i+1,:) = betarnd(a,b);

end

Markov\_chain\_1 = figure;

set(gcf,'Unit','Normalized','Position',[0.3 0.3 0.6 0.4]);

plot([1:ndraws]',X);

xlabel('Time');ylabel('X');

legend('chain1','chain2','chain3','chain4','chain5','chain6','Location','Best');

Title('Markov Chains');

We notice that the chains are initially in a state close to the starting state: this period is called the *burnin*. An important property of Markov chains is that, regardless of where we started, after this *burnin* period the chains reach a *steady state* in which the *states reflect samples* from a *stationary distribution*. The goal of MCMC is to design a Markov chain such that the stationary distribution of the chain is exactly the distribution that we are interesting in sampling from. This is called the *target distribution*. In other words, we would like the states sampled by a Markov chain to correspond to samples drawn from the target distribution. The idea is to set up the transition function such that regardless of how we initialize each chain, it will converge to the target distribution. Here we will discuss three forms of MCMC: *Metropolis, Metropolis-Hastings* and *Gibbs* sampling.

Suppose our goal is to sample from the target density *p*(*θ*). The *Metropolis sampler* creates a Markov chain that produces a sequence of values:



where represents the state of the Markov chain at iteration *t*. Thus, we want that, after the *burnin* period, samples from the chain reflect samples from the target distribution *p*(*θ*).

We start with an initial state . Since we want to sample from *p*(*θ*), we want to *move* along *θ* and accumulate more states *θ*. Our move from to is based on a *proposal distribution*. For example, we can generate a tentative move from a uniform distribution centered on the current state , or from a distribution with the mean at and controlling the proposed steps. The illustration on the side shows how this process works: 1. the current state of the chain is ; 2. a normal distribution around the current state is used to generate a proposal ; 3. *p*() is calculated; 4. based on the ratio *p*()/*p*() the proposal is accepted and the new state is set equal to the proposal ; the proposal distribution now centers on the state.

As an example of the *MCMC* method, let’s imagine our goal is to sample from a *Normal distribution* with . The *pdf* for the normal distribution is:

where is a normalizing constant. Since, as discussed below, the Metropolis sampler does not need normalizing constants the unnormalized *Normal* can be simplified to:

mu = 5;

sigma = 1;

gaussian = @(theta) exp( -(theta-mu).^2./(2\*sigma^2) );

In this and other examples of *MCMC* we use MATLAB code that was modified from the original written by *Mark Steyvers*, University of California, Irvine (<http://psiexp.ss.uci.edu/research/>).

Set initial parameters for the Metropolis sampler.

N = 30000; % Maximum number of states

thmin = -5; thmax = 15; % Range for starting values

th = zeros(1,N); % Initial storage space

rng('shuffle','twister');

th(1) = unifrnd(thmin,thmax); % Start value

Start sampling and iterate until a total of N states is accumulated.

delta = 0.5;

Proposal density

q = @(x,y) unifpdf(y-x,-delta,delta);

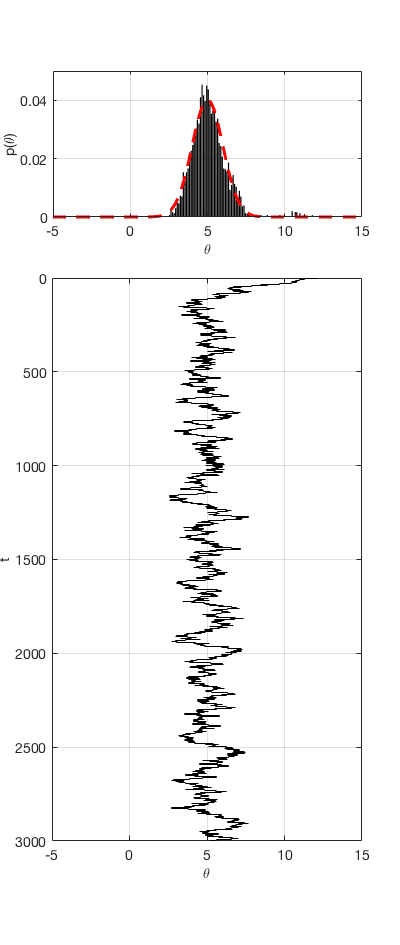
qrnd = @(x) x + rand\*2\*delta - delta;

t = 1;

while t < N

Sample from the proposal *uniform* density.

th\_star = qrnd(th(t));



Calculate the acceptance ratio:

alpha = min([1 gaussian(th\_star) / gaussian(th(t))]);

Draw a *uniform deviate* from [0 1].

u = rand;

Decide whether to accept or reject.

if alpha > u

th(t+1) = th\_star;

else

th(t+1) = th(t);

end

Increase the time count.

t = t + 1;

end

Display histogram and states history

Metropolis\_sampling = figure;

set(gcf,'Unit','normalized','Position',[0.1 0.1 0.2 0.9])

subplot(4,1,1);

h1 = histogram(th,'Normalization','pdf','FaceColor','k' );grid on

xlim([thmin thmax]);

xlabel('\theta'); ylabel('p(\theta)');grid on;hold on;

Overlay the theoretical density

theta\_range = thmin:.1:thmax;

y = normpdf(theta\_range,mu,sigma);

plot(theta\_range,y,'--r','LineWidth',3);

Display the states history (notice the *burnin* period from *t* = 0 to *t* ~ 150):

subplot(4,1,2:4);

stairs(th,1:N,'-k');

ylabel('t'); xlabel('\theta');

set(gca,'YDir','reverse');

xlim([thmin thmax]);grid on;

The following is a flowchart of the Metropolis method:

Initialize the first state, to some initial value.

⇓

Use a proposal distribution to generate a

candidate point

⇓

Evaluate an *acceptance probability*:

while *t<T*

⇓

Generate a uniform deviate *u*.

⇓

If accept the proposal and set

If reject the proposal and set

⇓

Set

The procedure continues until the sampler reaches convergence. At this point, the samples reflect samples from the target distribution *p*(*θ*). Since the Metropolis sampler involves only a ratio of densities, any terms independent of *θ* in *p*(*θ*) can be removed. Thus, it is not necessary to know the normalizing constant of the density or probability mass function. This feature of the Metropolis sampler is very important because sampling from unnormalized distributions happens frequently in Bayesian models, where calculating the normalization constant is often too difficult.

A requirement for the Metropolis sampler is that the proposal distribution must be symmetric, such that the probability of proposing a new state given the old state, is the same as proposing to go from the new state back to the old state:

This symmetry holds with proposal distributions such as, for example, *Normal*, *Cauchy*, *Student*-t, and *uniform*. If this symmetry does not hold, one must use the *Metropolis-Hastings (MH) sampler*, which is a generalization of the Metropolis method.

The *MH* iteration is essentially identical to the Metropolis iteration, the only difference being in the acceptance probability, which is defined as:

, which includes a *correction factor*

Initialize the first state, to some initial value.

⇓

Use a proposal distribution to generate a

candidate point

⇓

Evaluate an *acceptance probability*:

while *t<T*

⇓

Generate a uniform deviate *u*.

⇓

If accept the proposal and set

If reject the proposal and set

⇓

Set

The *MH* iteration is:

t = 1;

while t < T

th\_star = qrnd(th(t));

alpha = min( 1 , (gaussian(th\_star) / gaussian(th(t))) \* ...

q(th(t),th\_star)/q(th\_star,th(t)) );

u = rand;

if alpha > u

th(t+1) = th\_star;

else

th(t+1) = th(t);

end

t = t + 1;

end

The following is the implementation of the *MH* algorithm using MATLAB function *mhsample*:

N = 30000;

mu = 5;

sigma = 1;

rng('shuffle','twister');

thmin = -5; thmax = 15;

start = unifrnd(thmin,thmax);

gaussian = @(x) exp( -(x-mu).^2./(2\*sigma^2) );

delta = 0.5;

q = @(x,y) unifpdf(y-x,-delta,delta);

qrnd = @(x) x + rand\*2\*delta - delta;

th = mhsample(start,N,'pdf',gaussian,'proppdf',q,'proprnd',qrnd);

The *Metropolis* and *MH* sampler can be modified to work with multivariate (*N* variables) distributions. The first approach is to use *blockwise updates*. In this approach the proposal distribution *q* has the same dimensionality as the target distribution *p.* Specifically, if *p* is a distribution over *N* variables, then we must design a proposal distribution that is also a distribution involving *N* variables. We then accept or reject a proposed state bold x^*sampled from the proposal distribution *q* in exactly the same way as for the univariate Metropolis-Hastings algorithm.

However, when the number of dimensions *N* becomes large it becomes difficult to find a suitable proposal distribution with block-wise updates. One way to remedy this is to simply loop over the *N* dimensions bold xin sequence, sampling each dimension independently from the others. In this case, we design a 1-dimensional proposal distribution, and we accept or reject the proposal for each variable separately, keeping the other variables constant. This is what is known as using *component-wise updates*.

To understand how this works, we can look at the sampling of a bivariate distribution . After initializing the sampler with some suitable values , at each iteration *t*, we generate a proposal from , and we evaluate the acceptance ratio of against ; at this step we only vary the first component and keep the second component constant. If the proposal is accepted, is updated ( ). In the next step, we make a proposal from , and we evaluate the acceptance ratio of against . In this second step the first component is held constant to the updated value from the first step, which makes the second step conditional on the first step.

Initialize the first state, to some initial value.

⇓

**1.** Use a proposal distribution

to generate a

candidate point

⇓

Evaluate an *acceptance probability*:

⇓

Generate a uniform deviate *u*.

⇓

If accept the proposal and set

If reject the proposal and set

while *t<T*

⇓

**2.** Use a proposal distribution

to generate a

candidate point

⇓

Evaluate an *acceptance probability*:

⇓

Generate a uniform deviate *u*.

⇓

If accept the proposal and set

If reject the proposal and set

⇓

Set

As an example, let’s imagine our goal is to sample from a *Cauchy distribution* (CHAPTER 20, Special Topics). This is a continuous probability distribution also known as the Lorentz distribution, or the *Lorentzian* function, which is used in spectroscopy to describe the shape of spectral lines.

where is a normalizing constant. Since the Metropolis sampler does not need normalizing constants, the unnormalized Lorentzian can be simplified to:

cauchy = @(theta) 1 ./ (1 + theta.^2);

and the acceptance ratio is:

Metropolis sampling of bivariate Cauchy distribution.

clear, clc, close all

Initialize the Metropolis sampler.

N = 5000; % Maximum number of iterations

sigma = 1; % Standard deviation of the normal proposal density

thmin = [-30 -30]; % Minimum for theta1 and theta2

thmax = [30 30]; % Maximum for theta1 and theta2

th = zeros(N,2);

rng('shuffle','twister');

th(1,:) = unifrnd(thmin(1),thmax(1),1,2); % Start value for theta1

Start sampling

cauchy = @(theta) 1 ./ (1 + theta.^2);

t = 1;

while t < N % Iterate until we have N states

Propose a new value for theta1

th\_star = normrnd(th(t,1),sigma);

pratio = (cauchy(th\_star)\*cauchy(th(t,2)))/(cauchy(th(t,1))\*cauchy(th(t,2)));

alpha = min([ 1 pratio]);

Draw a uniform deviate from [ 0 1 ]

u = rand;

if alpha > u

th(t+1,1) = th\_star; % proposal becomes new value for theta1

end

Propose a new value for theta2

th\_star = normrnd(th(t,2),sigma);

pratio = (cauchy(th\_star)\*cauchy(th(t+1,1)))/(cauchy(th(t,2))\*cauchy(th(t+1,1)));

alpha = min([ 1 pratio]);

Draw a uniform deviate from [ 0 1 ]

u = rand;

if alpha > u

th(t+1,2) = th\_star; % proposal becomes new value for theta2

end

t = t + 1;

end

Histogram.

Bivariate\_Cauchy\_distribution = figure; clf;

subplot(1,2,1);

thbins1 = [thmin(1):thmax(1)];

thbins2 = [thmin(2):thmax(2)];

histogram2(th(:,1),th(:,2),'Normalization','pdf');

xlabel('\theta\_1'); ylabel('\theta\_2'); zlabel('pdf');

xlim([-30 30]);ylim([-30 30]);zlim([0 0.7]);box on

az = 61; el = 30;

view(az, el);

title('Histogram');

It’s worth noting here that the bivariate (or multivariate) Cauchy distribution is a special case of the bivariate (or multivariate) *t* distribution with degrees of freedom . The probability density function of the *d*-dimensional multivariate Student's *t* distribution is given by:

Thus, we can obtain the theoretical density using MATLAB function *mvtpdf*:

subplot(1,2,2);

Rho = [1 0; 0 1];nu = 1;

x1 = -30:30; x2 = -30:30;

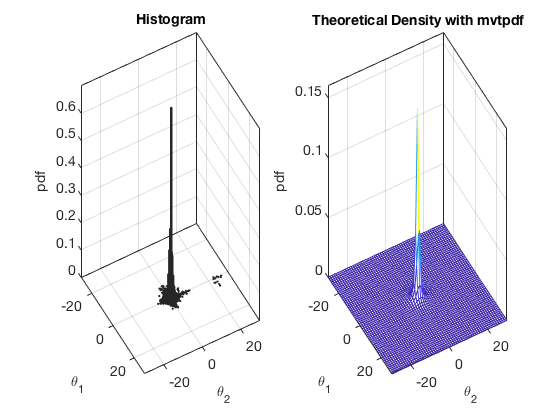
[X1,X2] = meshgrid(x1,x2);

F = mvtpdf([X1(:) X2(:)],Rho,nu);

F = reshape(F,length(x2),length(x1));

mesh(x1,x2,F);

caxis([min(F(:)),max(F(:))]);view(az, el);

axis([-30 30 -30 30 0 .16]);box on

xlabel('\theta\_1'); ylabel('\theta\_2');

zlabel('f(\theta\_1,\theta\_2)');;

title('Theoretical Density with mvtpdf');

A problem with the Metropolis-Hastings protocol is that a large number of samples are rejected. Component-wise updates for MCMC algorithms are generally more efficient for multivariate problems than blockwise updates, as it is more likely to accept a proposed sample by drawing each component/dimension independently from the others. However, samples may still be rejected too often, leading to excess computation.

The *Gibbs sampler*, another popular MCMC sampling technique, also uses component-wise updates, but unlike the Metropolis-Hastings algorithm, all proposed samples are accepted, so there is no wasted computation. An additional advantage is that there is no need to specify a proposal distribution. However, the Gibbs sampler is applicable only to certain classes of problems, based on two criteria:

1) it is necessary to have an *analytic* expression for the conditional distribution of each variable in the *target* joint distribution given all other variables in the joint. Formally, if the target distribution *p*(*X*) is *N*-dimensional, we must have *N* individual expressions for:

, which is the same as:



Each of these expressions defines the probability of the *i*-th dimension given that we have values for all other *j* dimensions. Having the conditional distribution for each variable means that we don’t need a proposal distribution or an accept/reject criterion. Therefore, we can simply sample from each conditional, while keeping all other variables fixed.

2) it must be possible to sample from each conditional distribution.

In some cases, these conditional distributions are not known, and Gibbs sampling cannot be applied. However, many Bayesian models use distributions that lend themselves easily to Gibbs sampling.

To illustrate the Gibbs sampler, we look at the case of bivariate normal distribution where we have a joint distribution . We first initialize the sampler with some suitable values for and . At each iteration *t*, we sample a new value for from a proposal . As opposed to Metropolis-Hastings, we always accept this proposal, so the new state is automatically updated. In the second step, we sample a new value for from the proposal . Therefore, the procedure involves iterative conditional sampling:

Initialize the first state, to some initial value.

⇓

Sample from

while *t<T*

⇓

Sample from

⇓

Set

For example, if we want to apply Gibbs sampling to a *normal* bivariate distribution:

we need to sample from the two conditional distributions, for which an *analytical* expression is known:

Parameters of the Bivariate normal

mu = [1.5 3];

sigma = [0.8 0.6;0.6 0.8];

Initialize the Gibbs sampler

N = 10000;

ndim = 2;

thmin = -3;thmax = 8;

rng('shuffle','twister');

dims = 1:ndim;

th = zeros(N,2);

th(1,:) = unifrnd(thmin,thmax,1,2);

Start sampling

t = 2;

while t < N

t\_id = [t-1,t];

Loop over dimensions

for i = 1:ndim

not current dimension

ni = dims~=i;

mu\_cond = mu(i) + sigma(ni,i)\*(th(t\_id(i),ni) - mu(ni));

std\_cond = sqrt(1 - sigma(ni,i)^2);

draw from the conditional distribution using MATLAB *normrnd*

th(t,i) = normrnd(mu\_cond,std\_cond);

end

t = t + 1;

end

Histogram

Bivariate\_normal\_distribution = figure; clf;

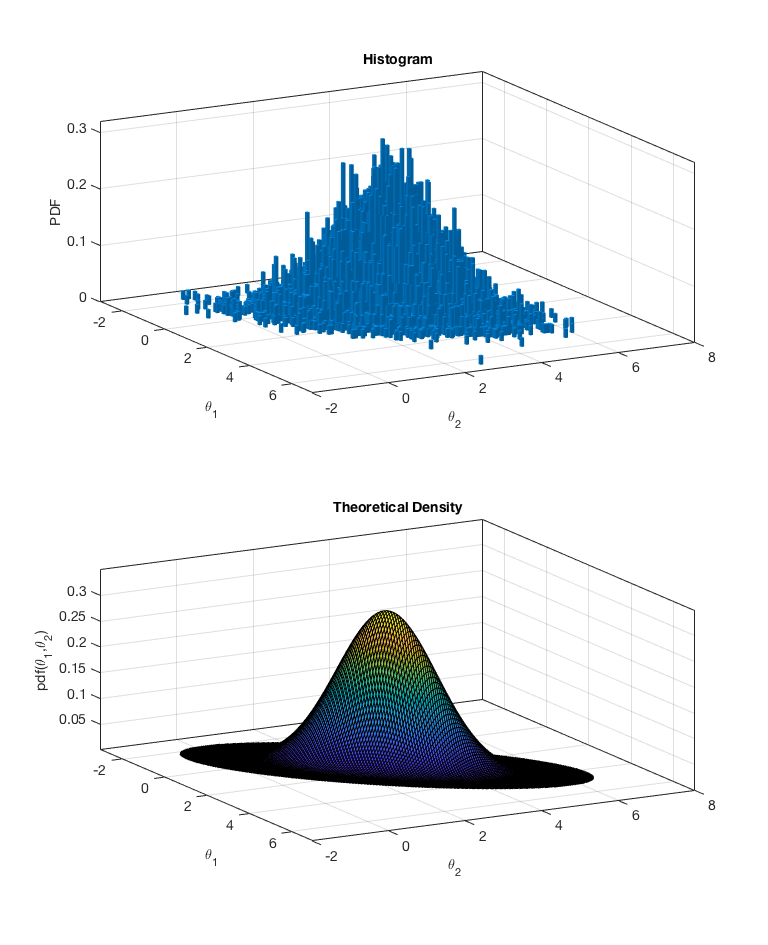
set(gcf,'Unit','Normalized','Position',[0.2 0 0.4 1]);

subplot(2,1,1);

thbins = thmin:.1:thmax;

nbins = length(thbins)

h12\_dens = histogram2(th(:,1),th(:,2),nbins,…

'Normalization','pdf')

xlabel('\theta\_1'); ylabel('\theta\_2');

zlabel('pdf');

xlim([thmin thmax]),ylim([thmin thmax]);

box on;az = 61; el = 30; view(az,el);

title('Histogram');

Theoretical density

subplot(2,1,2);

thbins = thmin:.05:thmax;

nbins = length(thbins);

[th1grid,th2grid] = meshgrid(thbins,thbins);

zgrid = mvnpdf([th1grid(:) th2grid(:)],mu,sigma);

zgrid = reshape(zgrid,nbins,nbins);

surf(thbins,thbins,zgrid);

xlabel('\theta\_1'); ylabel('\theta\_2');zlabel('pdf');

view(az,el); box on

xlim([thmin thmax]); ylim([thmin thmax]); zlim([1e-4 0.35])

title('Theoretical Density');

As a second example, we apply Gibbs sampling to a bivariate Cauchy distribution with the same mean and covariance matrix as the Normal distribution we have examined above. We recall here that the bivariate Cauchy is the bivariate *t* with :

we need to sample from the two conditional distributions, for which an *analytical* expression is known:

where:

and

where:

Or alternatively, ignoring a normalization factor:

where:

and

where:

However, because the conditional distribution of a bivariate Cauchy is itself a bivariate *t* distribution with , at each drawing of the distribution we obtain two coordinates (instead of just one), and we take their mean. To sample from the multivariate *t* distribution we use MATLAB *mvtrnd*:

Parameters

mu = [0 0]; sigma = [0.8 0.6; 0.6 0.8];

Initialize the Gibbs sampler

N = 20000;ndim = 2;

thmin = -5;thmax = 7;

rng('shuffle','twister');

dims = 1:ndim;

th = zeros(N,2); th(1,:) = unifrnd(thmin,thmax,1,2);

Start sampling from *t* multivariate distribution

t = 2;nu = 1;p = [2 2];

while t < N

Loop over dimensions

for i = 1:ndim

t\_id = [t-1,t];

not current dimension

ni = dims~=i;

mu\_cond = mu(i) + sigma(i,ni)\*(1/sigma(ni,ni))\*(th(t\_id(i),ni)-mu(ni));

d = (th(t\_id(i),ni)-mu(ni))\*(1/sigma(ni,ni))\*(th(t\_id(i),ni)-mu(ni))';

sigma\_ratio = (nu+d)/(nu+p(ni));

sigma\_cond = sigma\_ratio\*(sigma(i,i) - sigma(i,ni)\*(1/sigma(ni,ni))\*sigma(ni,i));

draw from conditional distribution

tsample = mvtrnd([sigma\_cond 0;0 sigma\_cond],nu+p(ni)) + mu\_cond;

th(t\_id(i),i) = mean(tsample,2);

end

t = t + 1;

end

Histogram

Bivariate\_cauchy\_distribution = figure; clf;

set(gcf,'Unit','Normalized','Position',[0.2 0 0.6 1]);

subplot(2,1,1);

thbins = thmin:.01:thmax;

nbins = length(thbins)

h12\_dens = histogram2(th(:,1),th(:,2),nbins,'Normalization','pdf')

xlabel('\theta\_1'); ylabel('\theta\_2'); zlabel('pdf');

xlim([thmin thmax]),ylim([thmin thmax]);box on

az = 61; el = 30; view(az,el);

title('Histogram from Gibbs sampling');

Theoretical density

subplot(2,1,2);

nu = 1;

x1 = thmin:.05:thmax; x2 = thmin:.1:thmax;

[X1,X2] = meshgrid(x1,x2);

F = mvtpdf([X1(:) X2(:)],sigma,nu);

F = reshape(F,length(x2),length(x1));

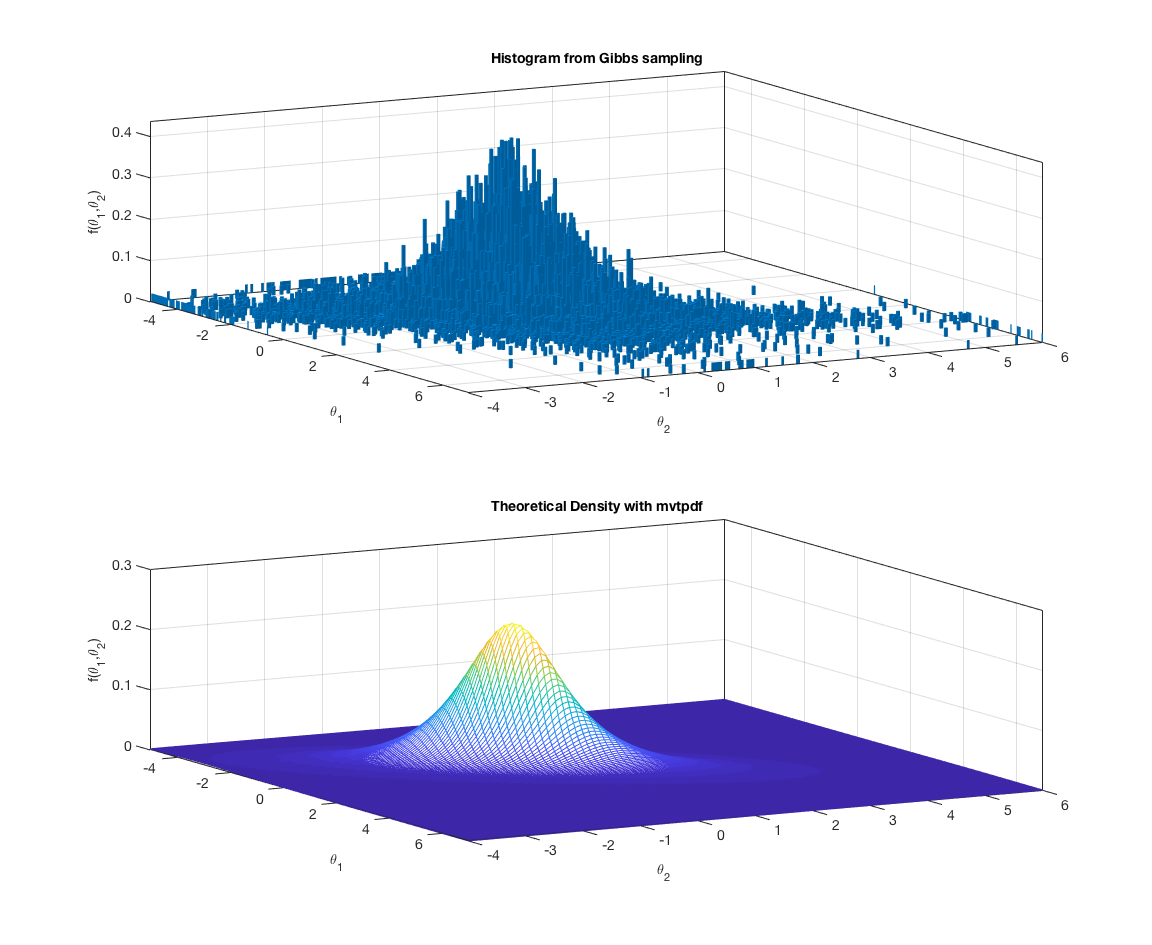
mesh(x1,x2,F);caxis([min(F(:)),max(F(:))]);

axis([thmin thmax thmin thmax 0 .30]);box on

view(az, el);

xlabel('\theta\_1'); ylabel('\theta\_2'); zlabel('pdf');

title('Theoretical Density with mvtpdf');



Application of MCMC to Bayesian data analysis.

There are a few basic rules that form the basis for all probabilistic approaches to solve inference problems:

1. *Sum rule*: we can rewrite the *marginal* probability of x as a sum over the joint distribution of x and y where we sum over all possibilities of y.

2. *Product rule*: we can rewrite a joint distribution as a product of a *conditional* and *marginal* probability.

3. *Bayes rule*: derived from the *product rule* by bringing in the equality,

on the other side:

and applying both *sum* and *product rule* to rewrite :

However, in inference techniques, there is rarely a need to evaluate explicitly the denominator , and the conditional probability in *Bayes rule* is evaluated only up to a constant of proportionality:

4. *Chain rule*: the product rule is applied repeatedly to give expressions for the joint probability involving more than two variables. For example, the joint distribution over three variables can be factorized into a product of conditional probabilities:

5. *Independence*: two variables are said to be independent if their joint distribution factorizes into a product of two marginal probabilities:

, wich implies:

6. *Conditional independence*: two variables *x* and *y* and said to be conditionally independent of *z* if the following holds:

, wich implies:

In other words, if we condition on *z*, and now also learn about *y*, this is not going to change the probability of *x*. However, *conditional independence* between *x* and *y* does not imply *independence* between *x* and *y*.

These rules are regularly applied in the development and application of probabilistic models: the goal is always to find the best parameter values of the model that explain the experimental observations.

Let’s imagine we have some experimental data *D*, and that we also have a model with parameters that is designed to explain the data in a generative sense (that is, it can simulate data that we can compare with the experimental one). This means that we can define the *likelihood* of the model, that is how likely is that we would reproduce the experimental data by running the generative model with some given values of the parameters . It is important to understand clearly the direction of this conditional probability:

The likelihood gives an explanation of the data in terms of the parameters .

The most basic approach to parameter estimation is *maximum likelihood (ML)*. In this approach we find the parameter estimates that *maximize* the likelihood of the data given a model that operates with those parameter estimates. This can be written as:

The ML estimate corresponds to the values of the parameters that best agree with the actually observed data: this is the *mode* of . However, it may not be easy to find the maximum of the likelihood function when there are many parameters in the model. In this case, numerical techniques that explore the gradient of the likelihood function might get stuck in suboptimal solutions (e.g., local modes).

The Bayesian approach to parameter estimation is to treat as a random variable, and to associate a *prior distribution* with it, which tells us how likely the parameter values are, *regardless* of any experimental data . The goal is to estimate the *posterior distribution* of the parameters given our data (*posterior inference*). Using Bayes rule, we can evaluate this conditional probability with:

As we have already mentioned, we rarely need to evaluate explicitly the denominator , and ignoring any normalizing constant we can write:

*posterior likelihood prior*

Thus, the posterior distribution is the product of the likelihood (how likely the data is given the model that uses these parameter values) and the prior (how likely these parameter values are to begin with).

One way to *fit* the model to the data is to find the *mode* of the posterior distribution: this is the set of the maximum a posteriori (*MAP*) parameter values:

One drawback with the *MAP* approach is that there could exist several different combinations of parameter values that give essentially identical high value of posterior probability, and is instead a *single* set of parameters. A more comprehensive approach consists in characterizing the *entire posterior distribution*, rather than just the mode. While in some cases it may be possible to find an analytic expression for this distribution, most often MCMC can be used to obtain samples from it.

For example, consider some *observed data* (evidence) represented by observations, for which we believe that each is an independent draw from a Normal distribution:

with and as *latent* *variables* (unobserved variables). The corresponding graphical model is:



The nodes corresponding to the *observations* are shaded in orange. The nodes corresponding to the *latent variables* are not shaded. The models in the left and right panels are identical, but the right panel provides a more compact representation (*plate notation*) of the repeated sampling steps.

Set mu and sigma

mu = 5; sigma = 1.5;

Calculate synthetic data

rng('Default')

y = normrnd(mu,sigma,1000,1);

*MLE* from normfit

[muhat,sigmahat] = normfit(y);

Having generate synthetic data, we can now consider and as unknown parameters that we want to estimate from this data. The *likelihood* of all observations is given by:

the *posterior* distribution is:

and we assume the *prior* distributions for and to be:

where are hyperparameters decided by us.

Given this structure for the *prior*  distributions, the *posterior* does not have a closed form, and we plan to use the *Metropolis-Hastings* or the *Gibbs* method to sample from it.

*Metropolis-Hastings algorithm*.

Model parameters

m = 0;s = 3;a = 2;b = 3;

Initialize the Metropolis-Hastings sampler

N = 5000; burnin = 100;

Starting value for muhat and sigmahat

muhat = unifrnd(0,10);sigmahat= unifrnd(0,4);

Sigma value for proposal normal distribution

pro\_sigma = 0.3;

Storage space for states and acceptance decisions

th = zeros(N,2); accept = zeros(N,2);

Start sampling

rng('shuffle','twister');

for t = 1:N

Propose a new muhat value from a normal proposal density

old\_muhat = muhat;

new\_muhat = normrnd(old\_muhat,pro\_sigma);

Calculate priors for new and old muhat

p\_newmu = normpdf(new\_muhat,m,s);

p\_oldmu = normpdf(old\_muhat,m,s);

Correcting factor (ratio of proposal densities)

p\_newmu\_g\_oldmu = normpdf(old\_muhat,pro\_sigma);

p\_oldmu\_g\_newmu = normpdf(new\_muhat,pro\_sigma);

proposal\_ratio = p\_oldmu\_g\_newmu / p\_newmu\_g\_oldmu;

Ratios for acceptance decision

prior\_ratio = p\_newmu / p\_oldmu;

likelihood\_ratio = prod(normpdf(y,new\_muhat,sigmahat) ./ ...

normpdf(y,old\_muhat,sigmahat));

posterior\_ratio = likelihood\_ratio \* prior\_ratio;

Acceptance decision

alpha = min([1 posterior\_ratio \* proposal\_ratio]);

u = rand;

if u < alpha

muhat = new\_muhat;

accept( t,1 ) = 1;

end

Propose a new sigmahat value from a normal proposal density

old\_sigmahat = sigmahat;

new\_sigmahat = normrnd(old\_sigmahat,pro\_sigma);

Calculate priors for new and hold sigmahat

p\_newsigmahat = sqrt(1/gampdf(new\_sigmahat,a,b));

p\_oldsigmahat = sqrt(1/gampdf(old\_sigmahat,a,b));

Correcting factor (ratio of proposal densities)

p\_newsigmahat\_g\_oldsigmahat = normpdf(old\_sigmahat,pro\_sigma);

p\_oldsigmahat\_g\_newsigmahat = normpdf(new\_sigmahat,pro\_sigma);

proposal\_ratio = p\_oldsigmahat\_g\_newsigmahat / ...

p\_newsigmahat\_g\_oldsigmahat;

Ratios for acceptance decision

prior\_ratio = p\_newsigmahat / p\_oldsigmahat;

likelihood\_ratio = prod(normpdf(y,mu,new\_sigmahat) ./ ...

normpdf(y,mu,old\_sigmahat));

posterior\_ratio = likelihood\_ratio \* prior\_ratio;

Acceptance decision

alpha = min( [ 1 posterior\_ratio \* proposal\_ratio ] );

u = rand;

if u < alpha

sigmahat = new\_sigmahat;

accept( t,2 ) = 1;

end

Store current states

th(t,:) = [muhat sigmahat];

end

Analyze every 'skip' states

skip = 5;

samples = th(burnin:skip:N,:);

Acceptance ratio for muhat and sigmahat

mean\_acceptance = mean(accept);

0.1914 0.1418

Calculate *posterior predictive* using the samples of muhat and sigmahat

nreps = 1000;

posterior\_pred\_samples = zeros(size(samples,1),nreps);

for reps = 1:nreps

posterior\_pred\_samples(:,reps) = normrnd(samples(:,1),samples(:,2));

end

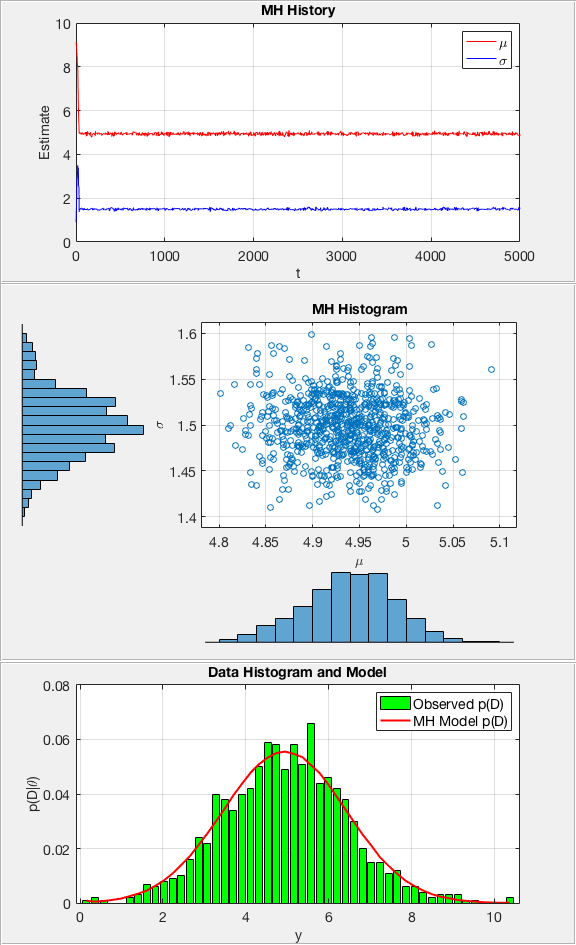
Means

mean\_mu\_sigma = mean(th);

std\_mu\_sigma = std(th);

ci\_mu\_sigma = [(mean\_mu\_sigma-1.96\*std\_mu\_sigma)' ...

(mean\_mu\_sigma +1.96\*std\_mu\_sigma)'];

Mean mu (MH): 4.938

Mean sigma (MH): 1.499

CI mu (MH): 4.845 5.032

CI sigma (MH): 1.432 1.565

Display states across all iterations

Metropolis\_Hastings = figure;

set(gcf,'Unit','Normalized','Position',[0.1 0 0.3 0.9])

hp1 = uipanel('position',[0 0.7 1 0.3])

axes('Parent',hp1)

plot(1:N,th(:,1),'-r',1:N,th(:,2),'-b');

xlabel('t'); ylabel('Estimate');grid on

legend('\mu','\sigma');

title('MH History')

Display scatter plot of the samples

hp2 = uipanel('position',[0 0.3 1 0.4])

axes('Parent',hp2)

scatterhist(samples(:,1),samples(:,2));grid on

xlabel('\mu');

ylabel('\sigma');

title('MH Histogram')

Display data histogram and superimpose posterior predictive

hp3 = uipanel('position',[0 0.0 1 0.3])

axes('Parent',hp3)

h1 = …

histogram(y,'Normalization','pdf','FaceColor','g')

hold on

h2 = histogram(posterior\_pred\_samples(:),'Normalization','pdf',...

'FaceColor','none','EdgeColor','none')

h3 = plot(h2.BinEdges(2:end),h2.Values,'-r','LineWidth',1.5);grid on

xlabel('y');ylabel('p(D|\theta)');

legend([h1 h3],'Observed p(D)','MH Model p(D)')

title('Data Histogram and Model')

*Gibbs algorithm*.

Using Bayestheorem it can be shown that the *conditional* distributions for , and for are:

Where are defined with respect to the hyperparameters used in the MH algorithm:

Model parameters

m = 0;s = 3;s2 = s^2;a = 2;b = 3;

n = size(y,1);ytot=sum(y);

Allocate arrays

N = 5000;burnin = 50;

Starting value for mu and sigma

mu = zeros(N,1);mu(1) = unifrnd(0,10);

sigma = zeros(N,1);sigma(1) = unifrnd(0,4);

Start Gibbs Sampler

for t=2:N

Draw for mu

sigma2 = sigma(t-1)^2;

s2star = 1/(1/s2+n/sigma2);

mstar = s2star\*(m/s2 + ytot/sigma2);

mu(t) = normrnd(mstar,sqrt(s2star));

Draw for sigma2

astar = a+n/2;

bstar = b+sum((y-mu(t)).^2)/2;

sigma(t) = sqrt(1/gamrnd(astar,1/bstar));

end

Analyze every 'skip' states

skip = 5;

samples = th(burnin:skip:N,:);

Calculate posterior predictive using the samples of mu and sigma

nreps = 1000;

posterior\_pred\_samples = zeros(size(samples,1),nreps);

for reps = 1:nreps

posterior\_pred\_samples(:,reps) = normrnd(samples(:,1),samples(:,2));

end

Means

mean\_mu\_sigma = mean(th);

std\_mu\_sigma = std(th);

ci\_mu\_sigma = …

[(mean\_mu\_sigma-1.96\*std\_mu\_sigma)' ...

(mean\_mu\_sigma +1.96\*std\_mu\_sigma)'];

Display states across all iterations

Gibbs = figure;

set(gcf,'Unit','Normalized','Position',[0.1 0 0.3 0.9])

hp1 = uipanel('position',[0 0.7 1 0.3])

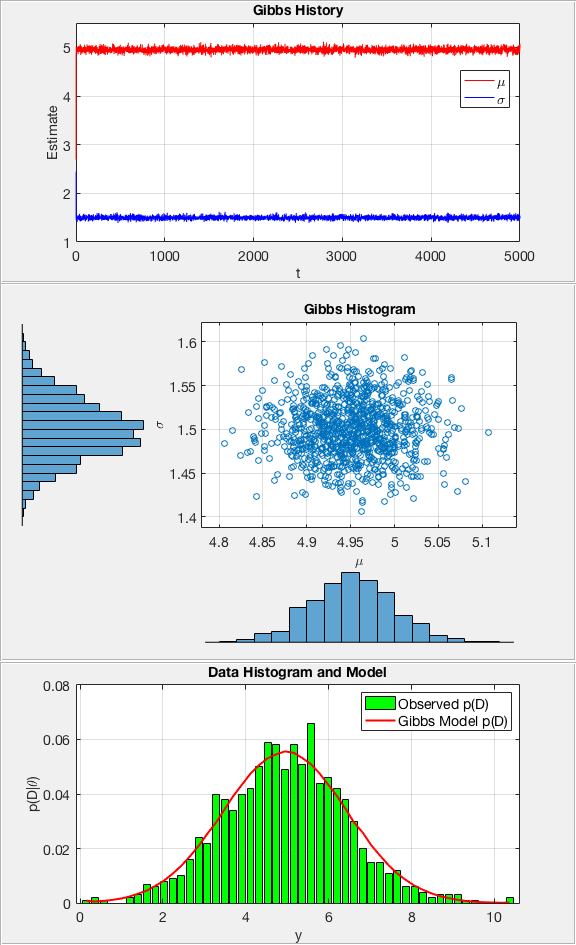
axes('Parent',hp1)

plot(1:N,th(:,1),'-r',1:N,th(:,2),'-b');

xlabel('t'); ylabel('Estimate');grid on

legend('\mu','\sigma');

title('Gibs History')



Display scatter plot of the samples

hp2 = uipanel('position',[0 0.3 1 0.4])

axes('Parent',hp2)

scatterhist(samples(:,1),samples(:,2));grid on

xlabel('\mu');

ylabel('\sigma');

title('Gibbs Histogram')

Display data histogram and superimpose posterior predictive

hp3 = uipanel('position',[0 0.0 1 0.3])

axes('Parent',hp3)

h1 = histogram(y,'Normalization','pdf','FaceColor','g');hold on

h2 = histogram(posterior\_pred\_samples(:),'Normalization','pdf',...

'FaceColor','none','EdgeColor','none')

h3 = plot(h2.BinEdges(2:end),h2.Values,'-r','LineWidth',1.5);grid on

xlabel('y');ylabel('p(D|\theta)');

legend([h1 h3],'Observed p(D)','Gibbs Model p(D)')

title('Data Histogram and Model')

In the case of this particular data set the *MLE* and *MCMC* model estimates of and of the parameters and are almost identical:

|  |  |  |  |
| --- | --- | --- | --- |
|  | MLE | MCMC-MH | MCMC-Gibbs |
| Mean | 4.9511 | 4.938 | 4.951 |
| Mean | 1.4984 | 1.499 | 1.500 |
| CI | 4.858-5.044 | 4.845-5.032 | 4.859-5.043 |
| CI | 1.436-1.567 | 1.432-1.565 | 1.435-1.565 |

**SPECIAL TOPICS: Hidden Markov Models**

A standard approach to probabilistic modeling with states evolving over time is to use *hidden Markov models (HMMs)*. The main difference between Markov models and *HMMs* is that the *state* of the system in a *HMM* is hidden (i.e., it is a *latent variable*). Thus, a *hidden Markov model* (HMM) is one in which you observe a sequence of *emissions*, but do not know the sequence of *states* the model went through to generate the emissions. Analysis of a hidden Markov model seeks to recover the sequence of states from the observed data.

A graphical model of an HMM is shown on the side. The variable *xt* represents the latent system state of the system at time *t.* The variable *yt* represents the observed set of measurements at time *t*. The system *state* evolves over time according to *p*(*xt* | *xt-1*), a standard first order Markov process where the state of system at time *t* depends only on the state at time *t-1*. The measurements (*emissions*) *yt* at time *t* are determined by *p*(*yt* | *xt*). Therefore, the observations at *yt* are determined solely by the state of the system at time *t-1*. The *HMM* model is completed by specifying a *prior* *x0* over the states at the start of the measurements.

As an example (taken from MATLAB manual), consider a simple Markov model with 2 hidden states representing the emotional state of a person, “Happy” or “Sad”, and 2 possible emissions, “Smile” or “Grimace”, representing the manifestation of these states on the person’s face. The probability that either emotional hidden state changes is *a=0.1*, and the probability that the state remains the same is (*1 – a)=0.9*. We also need to specify a prior *p*(x0) over states at the first time step which we assume to be uniform. The model is parameterized in such a way that the person produces an outcome that is consistent with the emotional state with probability *b=0.8*: thus, smiles are more likely in a happy state and grimaces more likely in a sad state. We represent “Sad” and “Grimace” with number 1, and “Happy” and “Smile” with number 2.

This Markov chain is defined by the *transition matrix* (*Markov matrix*, *right stochastic matrix*: see SPECIAL TOPICS, below):

and the *emission matrix* :

in which the 1st and 2nd row of the matrix refer to the 1st and 2nd emotional state, and the 1st and 2nd row of the matrix refer to the face expressions produced by the corresponding states.

Based on this *HMM* we can generate synthetic data using a simplified version of MATLAB function *hmmgenerate*. As it is, the model is not hidden because we are generating both the sequence of hidden emotional states and of emitted face expressions. However, if someone else was generating the emissions without telling us about the emotional states, all we would see are the face expressions: if we see a smile, we suspect the person is “Happy”, but we can’t be absolutely sure.

We starts the model in state 1 at time *t* = 1 and then make a transition to the first step using the probabilities in the first row of the transition matrix. So in the example above, at time *t* = 2 the probability of state 1 to remain state 1 is 0.9 and that of changing to state 2 is 0.1. If at *t* = 2 the state changes to 2, then at *t* = 3 the probability of state 2 to remain state 2 is 0.1 and that of changing to state 1 is 0.9. Similar considerations apply to the emission matrix.

N = 400;

T = [0.9 0.1;0.1 0.90];

E = [0.8 0.2;0.2 0.8];

Number of states and emissions

numStates = size(T,1);

numEmissions = size(E,2);

Create two random sequences, one for state changes, one for emission

statechange = rand(1,N);

emissionchange = rand(1,N);

Calculate cumulative probabilities

Tc = cumsum(T,2);

Ec = cumsum(E,2);

Normalize

Tc = Tc./repmat(Tc(:,end),1,numStates);

Ec = Ec./repmat(Ec(:,end),1,numEmissions);

States and emission possible values

states = 1:numStates;

emissions = 1:numEmissions;

State and emission sequences

state = zeros(N,1);

emission = zeros(N,1);

state(1) = states(1);

emission(1) = emissions(1);

for i = 2:N

state(i) = find(Tc(state(i-1),:)>=statechange(i),1);

emission(i) = find(Ec(state(i),:)>=emissionchange(i),1);

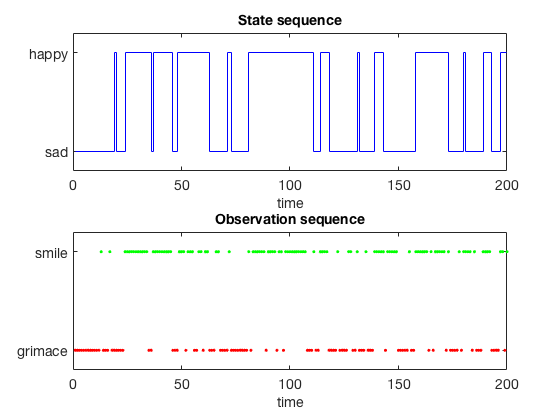
end

Plots

HMM = figure;

subplot(2,1,1)

stairs(state,'-b');ylim([0.8 2.2]);

title('State sequence');xlabel('time');

set(gca,'YTick',[1 2],'YTickLabel',…

{'sad','happy'});

subplot(2,1,2)

ind1 = find(emission == 1);

ind2 = find(emission == 2);

plot(ind2,emission(ind2),'.g',…

'MarkerSize',6);

ylim([0.8 2.2]);hold on

plot(ind1,emission(ind1),'.r',…

'MarkerSize',6);

ylim([0.8 2.2]);hold on

title('Observation sequence');

xlabel('time');

set(gca,'YTick',[1 2],'YTickLabel',{'grimace','smile'});

Matlab provides a set of functions to generate and analyze *HMM*  models. Thus, the same (but with different random number generation) synthetic data can be obtained with the following syntax:

Synthetic data

N = 400;

T = [0.9 0.1;0.1 0.90];

E = [0.8 0.2;0.2 0.8];

[seq,states] = hmmgenerate(N,T,E);

state\_vec = [states' seq'];

Here we derive the maximum likelihood estimate of ***T*** and ***E*** if we know the states:

[T\_inf, E\_inf] = hmmestimate(seq,states);

Here we calculate the *posterior state probabilities*, PSTATES, of the observed sequence of emissions, given ***T*** and ***E***. The posterior state probabilities are the conditional probabilities of being at state *k* at step *i*, given the observed sequence of emissions.

PSTATES = hmmdecode(seq,T,E);

Here, given a sequence of emissions, we use the *Viterbi* algorithm to calculate the most likely path through the hidden Markov model specified by the transition probability matrix, ***T***, and the emission probability matrix, ***E***. The Viterbi algorithm is a maximum likelihood estimator of the posterior, thus it only gives the mode of the posterior distribution over state sequences but it does not give the full posterior distribution or samples from the posterior distribution as with MCMC methods.

STATES = hmmviterbi(seq,T,E);

HMM = figure;

set(gcf,'Unit','normalized','Position',[0.2 0.1 0.6 0.9]);

subplot(5,1,1)

stairs(states,'-b');ylim([0.8 2.2]);

title('State sequence');xlabel('time');

set(gca,'YTick',[1 2],'YTickLabel',{'sad','happy'});

subplot(5,1,2)

seq1\_ind = find(seq == 1);

seq2\_ind = find(seq == 2);

plot(seq2\_ind,seq(seq2\_ind),'.g','MarkerSize',6);ylim([0.8 2.2]);hold on

plot(seq1\_ind,seq(seq1\_ind),'.r','MarkerSize',6);ylim([0.8 2.2]);hold on

title('Observation sequence');xlabel('time');

set(gca,'YTick',[1 2],'YTickLabel',{'grimace','smile'});

subplot(5,1,3)

stairs(STATES,'-b');ylim([0.8 2.2]);hold on

plot(PSTATES(2,:)+1,'--g')

title('State sequence (Viterbi) , State "happy" probability');xlabel('time');

set(gca,'YTick',[1 2],'YTickLabel',{'sad','happy'});

subplot(5,1,4)

PSTATES = hmmdecode(seq,T\_inf,T\_inf);

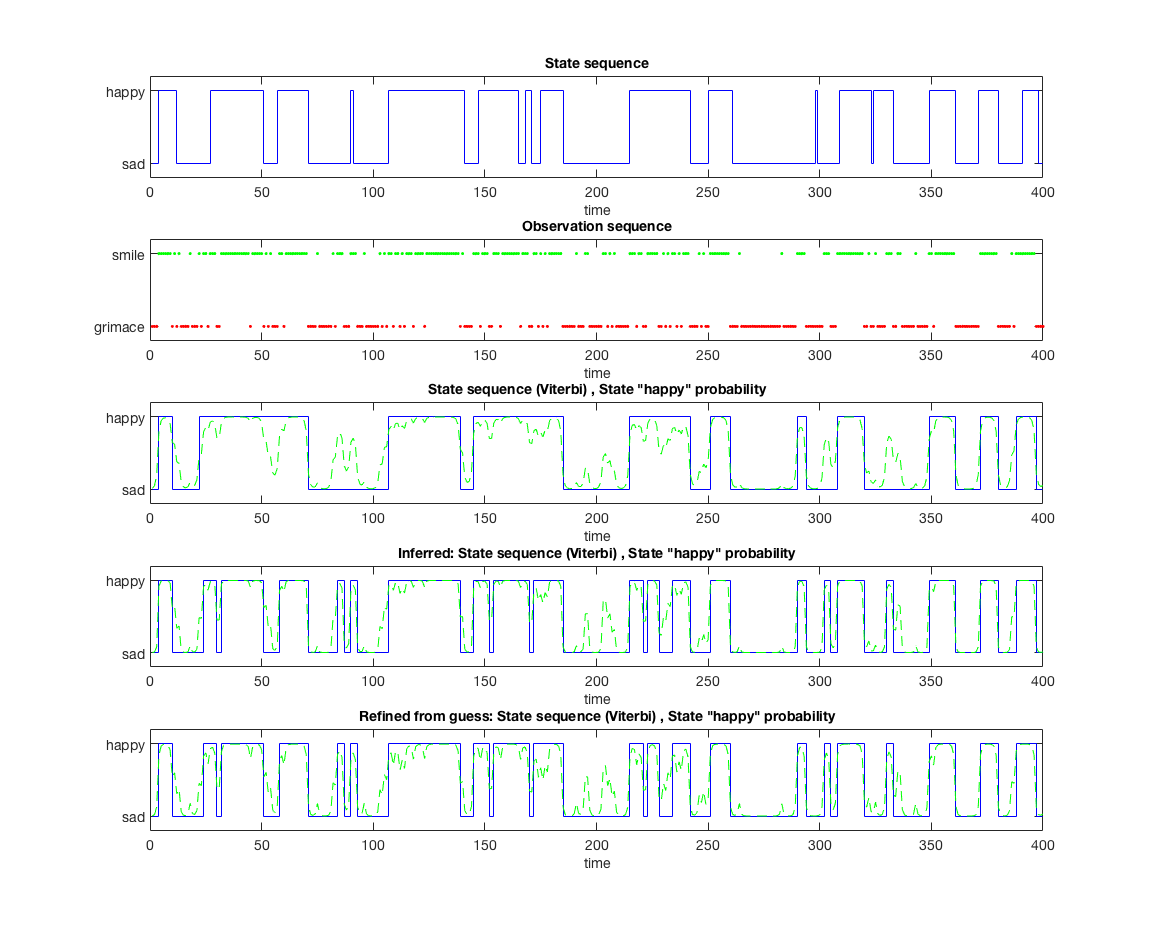
STATES = hmmviterbi(seq,T\_inf,T\_inf);

stairs(STATES,'-b');ylim([0.8 2.2]);hold on

plot(PSTATES(2,:)+1,'--g')

title('Inferred: State sequence (Viterbi) , State "happy" probability');xlabel('time');

set(gca,'YTick',[1 2],'YTickLabel',{'sad','happy'});



Here we estimates the transition and emission probabilities for a hidden Markov model using the *Baum-Welch* algorithm.  ***T\_guess*** and ***E\_guess***are initial estimates of the transition and emission probability matrices.

T\_guess = [0.7 0.3;

0.2 0.80];

E\_guess = [0.6 0.4;

0.3 0.7];

[T\_est,E\_est] = hmmtrain(seq,T\_guess,E\_guess);

subplot(5,1,5)

PSTATES = hmmdecode(seq,T\_est,T\_est);

STATES = hmmviterbi(seq,T\_est,T\_est);

stairs(STATES,'-b');ylim([0.8 2.2]);hold on

plot(PSTATES(2,:)+1,'--g')

title('Refined from guess: State sequence (Viterbi) , State "happy" probability');xlabel('time');

set(gca,'YTick',[1 2],'YTickLabel',{'sad','happy'});

**SPECIAL TOPICS: Markov matrices.**

A Markov matrix (also termed *stochastic* matrix, probability matrix, transition matrix, substitution matrix) is a matrix used to describe the transitions of a *Markov chain*. Each of its entries is a nonnegative real number representing a probability. There are three types of stochastic matrices:

A *right stochastic matrix* is a real square matrix, with each row summing to 1.

A *left stochastic matrix* is a real square matrix, with each column summing to 1.

A *doubly stochastic matrix* is a square matrix of nonnegative real numbers with each row and column summing to 1.

Accordingly, we define a *stochastic vector* (or *probability vector*) any vector whose elements are nonnegative real numbers which sum to 1. Thus, each row of a right stochastic matrix (or column of a left stochastic matrix) is a stochastic vector.

The key property of a stochastic matrix ***T*** is that its largest *eigenvalue* λmax = 1, and its corresponding *eigenvector* is real and positive. It’s straightforward to recognize that if ***T*** is a Markov matrix then λmax = 1. We recall here that if then:

and the eigenvectors ***x*** are in the *null space* of . If has non zero solution (the *null space* is not empty), then is not invertible and its determinant must be zero. Now, we know that in a *left stochastic matrix* each column of sums up to 1, and thus every column of adds up to 0.For example, if:

which means the rows of add up to 0, and thus they are linearly dependent. Therefore, is singular, which means:

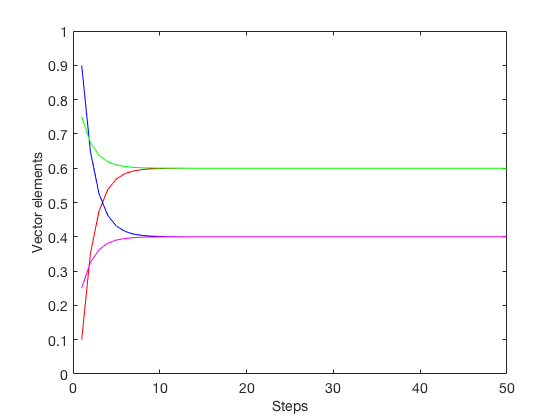
since we also have:

then one must be 1. But also no other can be larger than 1. In fact, if that was so then the powers would grow, while it’s easy to confirm that any power of a Markov matrix is still a Markov matrix:

, and so on …

Thus, λmax = 1controls the powers of ***T***. Similar proofs can be derived also for *right stochastic matrices* and *doubly stochastic matrices*.

If we multiply a positive vector ***u0*** = (a,1-a) by the left stochastic matrix ***T*** after *k* steps we have ***uk*** *=* ***T****k****u0***. After an infinite number of steps ***u∞*** is always the same regardless of the starting value ***u0***. For example, using even just 50 steps:

T = [.8 .3;.2 .7];

u0\_1 = [.1;.9];u0\_2 = [.75;.25];

npowers = 50;

u1 = zeros(2,npowers)

u2 = zeros(2,npowers)

u1(:,1) = u0\_1; u2(:,1) = u0\_2;

for i = 2:npowers

u1(:,i) = T\*u1(:,i-1);u2(:,i) = T\*u2(:,i-1);

end

Markov\_matrix\_1 = figure;

plot(1:npowers,u1(1,:),'-r',1:npowers,u1(2,:),'-b',...

1:npowers,u2(1,:),'-g',1:npowers,u2(2,:),'-m');

ylim([0 1])

xlabel('Steps');ylabel('Vector elements')

Since the columns of ***T*** sum up to 1 and the elements of also sum up to 1, the elements of any ***uk*** *=* ***T****k****u0*** also sum up to 1. We can easily derive the matrix by recalling that:

For the matrix:

and therefore:

Since is the same regardless of the initial value , this also means that the following *steady state* equation holds:

and the *steady state* is an eigenvector of ***T*** with eigenvalue 1. In fact:

[S,D] = eig(T)

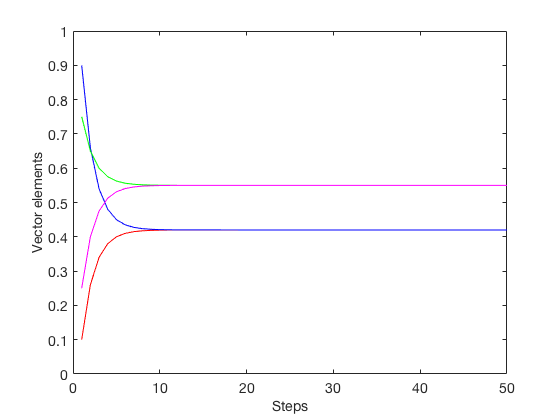
D\_inf = [1 0;0 0]

T\_inf = S\*D\_inf/S

Here we scale the eigenvector with eigenvalue 1 so that its elements sum up to 1.

u\_inf = S(:,1)/sum(S(:,1))

Notice that the situation is unchanged if we have a *right stochastic matrix* rather than a *left stochastic matrix*.

T = [.8 .3;.2 .7]';

u0\_1 = [.1;.9];u0\_2 = [.75;.25];

npowers = 50;

u1 = zeros(2,npowers);u2 = zeros(2,npowers)

u1(:,1) = u0\_1;u2(:,1) = u0\_2;

for i = 2:npowers

u1(:,i) = T\*u1(:,i-1);

u2(:,i) = T\*u2(:,i-1);

end

Markov\_matrix\_2 = figure;

plot(1:npowers,u1(1,:),'-r',1:npowers,u1(2,:),'-b',...

1:npowers,u2(1,:),'-g',1:npowers,u2(2,:),'-m');

ylim([0 1])

xlabel('Steps');ylabel('Vector elements')

[S,D,W] = eig(T)

u\_inf = S(:,1)/sum(S(:,1))

D\_inf = [1 0;0 0]

T\_inf = S\*D\_inf/S

,

or using the left eigenvectors :

**SPECIAL TOPICS: Statistical Hypothesis Testing, Bootstrap, Jackknife, Cross-validation.**

Statistical Hypothesis Testing.

In hypothesis testing, we start with a statistical hypothesis, about one or more populations. For example, we might want to determine whether a new formulation of a popular analgesic for migraine acts more rapidly at equal dosage than an older one.

We generally formulate our statistical hypotheses in two parts. The first is the *null hypothesis*, *H0*. It is *null* in the sense that it lacks an effect. Using the example above, we would define as the *null hypothesis* that the new formulation does not change the mean wait time before effective relief from migraine.

We also need an alternative hypothesis *H1* (i.e., the formulation does change the mean wait time). If we reject *H0*, it means we are accepting *H1*.

Typically, hypothesis testing requires the following steps:

1. Take a random sample from the relevant population. For example, we ask 100 patients suffering from recurring migraines to record at the next episode the time they felt effective relief after taking the new formulation.

2. Determine the *null* and *alternative* hypotheses.

Previous extensive testing with the old formulation has yielded a value for the population mean minutes with a standard deviation minutes.

The *null* hypothesis is that the population mean, as inferred from the sample mean, has not changed:

*H0*, .

The *alternative* hypothesis is that the population mean has changed:

*H1*,

or more precisely that the population mean has shortened:

*H1*,

From the patients’ report about the new formulation, we calculate a sample mean with standard deviation minutes.

Again, we test the *null* hypothesis *H0* that the sampled data comes from a normal distribution with population mean and standard deviation against the alternative hypothesis *H1* that the population mean is .

3. Calculate a *statistic* (this is called a *test statistic*) from the sample that provides information about the null hypothesis. A statistic is itself a random variable and has a probability distribution associated with it.

In order to decide whether or not an observed value of the statistic is consistent with the null hypothesis, we must know the distribution of the statistic when the null hypothesis is true.

As long as the sample size *n* is large enough, either a *z*-test or a *t*-test can be carried out in this case. Both tests use the sample mean . The difference between the two tests is that a *z*-test assumes the standard deviation  of the underlying normal distributionis known, while a *t*-test does not. As a result, a *t*-test must compute an estimate of the standard deviation directly from the sample. Thus, in our case the two tests yield, respectively:

where and represent the standard deviation of the mean.

x\_mean = 35; x\_sigma = 17

mu\_0 = 40; sigma = 15

n = 100

z = (x\_mean - mu\_0)/(sigma/sqrt(n))

t = (x\_mean - mu\_0)/(x\_sigma/sqrt(n))

4. Accept or reject the *null* hypothesis.

* If the value of the statistic is consistent with *H0*, then we accept it.
* If the value of the statistic is not consistent with *H0*, then we reject *H0* and accept *H1*.

Thus, in our case if the null hypothesis is really true, we have that the value of the sample mean, 35, is ~3.3 standard deviations away from the population mean (according to the *z*-test). The *z*-statistic has a standard normal distribution, *N*(0,1), and we know that approximately 95% of normally distributed random variables fall within two standard deviations on either side of the mean. We notice the following:

• If a positive value of the test statistic provide evidence for the alternative hypothesis, then the *critical region* (the interval of values for the test statistic over which we reject *H0*) is in the *upper tail* of the distribution of the test statistic. This is referred to as an *upper tail test*.

• If a negative value of the test statistic provides evidence for the alternative hypothesis, then the critical region is in the *lower tail* of the distribution of the test statistic. This is referred to as a *lower tail test*.

• If positive or negative values of the test statistic indicate evidence for the alternative hypothesis, then the critical region is in the *lower and upper tails*. This is referred to as a *two-tail test*.

We can determine exactly where the calculated *z* value falls in the standard normal distribution, *N*(0,1), and the calculated *t* value falls in the Student's *t* distribution with n – 1 degrees of freedom. For small sample sizes, Student's *t* distribution is flatter and wider than *N*(0,1), compensating for the decreased confidence in the estimate . As sample size increases, however, Student's *t* distribution approaches the standard normal distribution, and the two tests become essentially equivalent.

z\_t\_test = figure;

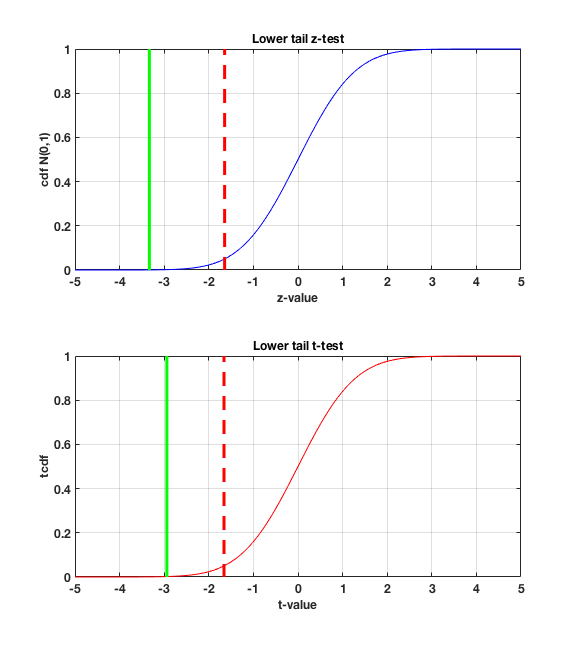
set(gcf,'Unit','Normalized','Position',…

[0.2 0.6 0.3 0.6]);

subplot(2,1,1)

plot(xvec,yvec\_z,'-b')

ylabel('cdf N(0,1)')

vline(z,{':g', 'LineWidth', 5})

title('Lower tail z-test')

subplot(2,1,2)

plot(xvec,yvec\_t,'-r')

ylabel('tcdf')

vline(t,{':g', 'LineWidth', 5})

title('Lower tail t-test')

In hypothesis testing, we typically use a quantity called a *p*-value. A *p*-value is defined as the probability of observing a value of the test statistic as extreme or more extreme than the one that is observed, when the null hypothesis is true. The word extreme refers to the direction of the alternative hypothesis. For example, if a small value of the test statistic (a lower tail test) indicates evidence for the alternative hypothesis, then the *p*-value is calculated as:

p\_z = normcdf(z)

p\_t = tcdf(z,n-1)

p\_z = 4.2906e-04

p\_t = 6.0400e-04

There are two types of errors that can occur when we make a decision in statistical hypothesis testing. The first is a *Type I error*, which occurs when we reject an *H0* that is true. The other error is called *Type II error*, which occurs when we fail to detect that *H0* is false.

|  |  |  |
| --- | --- | --- |
| Type of Error | Description | Probability of Error |
| Type I | we reject an *H0* that is true |  |
| Type II | we fail to detect that *H0* is false |  |

We are usually searching for significant evidence that the alternative hypothesis is valid, and we do not want to change from the *status quo* (i.e., reject *H0*) unless there is *clear* evidence in the data to lead in that direction. So, when setting up a hypothesis test we ensure that the probability of wrongly rejecting is controlled.

The probability of making a Type I error is denoted by  and is called the *significance level*of the test. This level is set by the analyst, and it represents the maximum probability of Type I error that will be tolerated. Typical values of  are 0.01 or 0.05.

The *critical value* (the value of the test statistic that divides the domain of the test statistic into a region where will be *H0* rejected and one where it will be accepted) is found as the *quantile* (inverse cdf) that gives a significance level of .

z\_critical\_value = norminv(0.05,0,1)

t\_critical\_value = tinv(0.05,n-1)

figure(z\_t\_test)

subplot(2,1,1)

vline(z\_critical\_value,{'--r', 'LineWidth', 3})

subplot(2,1,2)

vline(t\_critical\_value,{'--r', 'LineWidth', 3})

In our example, a sample mean of 35 minutes corresponds to a *p*-value of 4.2906e-04 (*z* statistics) or 6.0400e-04(*t* statistics) (green vertical lines in the cdf figure above), which is much smaller than the tolerance value  of a Type I error. Notice how the green line is far to the left than the red line, which represents the *critical value*. Thus, a sample mean of 35 minutesis inconsistent with the null hypothesis.

Monte Carlo Methods in hypothesis testing.

For many statistics the sampling distribution is known. However, in many cases, we do not know the sampling distribution for the statistic, but we are willing to make assumptions about the distribution.

The fundamental idea behind Monte Carlo simulation is to estimate the distribution of the statistic by randomly sampling from the population and recording the value of the statistic for each sample. The observed values of the statistic for these samples are used to estimate the distribution.

The first step is to identify a pseudo-population that can provide a good representation of the real population in all relevant aspects. The word pseudo-population refers to the fact that we obtain our samples using pseudo-random numbers. For example, if the random variable represents the time before a mechanical component of an engine fails, we can assume that the underlying population is exponentially distributed.

In the previous section we have seen how we can use the *significance level*  to find the appropriate *critical region* in the distribution of the test statistic when the null hypothesis is true. In the Monte Carlo method we sample randomly from the pseudo-population representing the null hypothesis, calculate the value of the test statistic at each trial, and use these values to estimate the distribution of the test statistic, and from it the critical value.

Here we use a simple example to illustrate the use of Monte Carlo methods in hypothesis testing. The mc\_data sample contains 30 observations. We are interested in using this sampe to test the following null and alternative hypotheses:

*H0*:

*H1*:

We use a *z* test statistics:

Load the sample.

load DATABASE/mc\_data

n = length(mc\_data);

The population mean and sigma are known.

mu\_0 = 395;

sigma = 7.1;

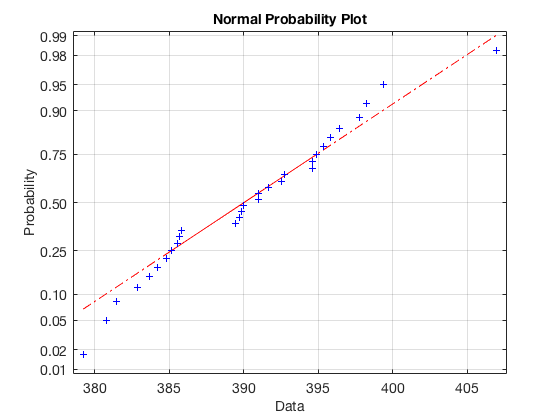
xbar = mean(mc\_data);

sigxbar = sigma/sqrt(n);

Here we get the *observed value* of the test statistic.

zobs = (xbar-mu\_0)/sigxbar;

*z*obs = -3.56

The next step is to decide which model to use for the population that generated the sample. We check whether a normal distribution with is a good model using a normal probability plot. We use Matlab plotting function *normplot*.

normplot(mc\_data)

box on

The resulting plot shows that we can use the normal distribution as the pseudo-population.

For the Monte Carlo simulation we use 1000 trials. At each trial, we randomly sample from the distribution of the test statistic under the null hypothesis (the normal distribution with and ) and record the value of the test statistic.

m = 1000;

zvec = zeros(m,1);

for i = 1:m

xs = normrnd(mu\_0,sigma,[n,1]);

zvec(i) = (mean(xs) - mu\_0)/sigxbar;

end

or simply:

xs = normrnd(mu\_0,sigma,[n,1000]);

zvec = (mean(xs) - mu\_0)/sigxbar;

Here we get the critical value for alpha by inverse cdf. This is a lower-tail test, so it is the alpha quantile.

alpha = 0.05;

cv = quantile(zvec,alpha)

cv = -1.6383

We get an estimated critical value of -1.64. Since the observed value of our test statistic is -3.56, which is less than the estimated critical value, we reject *H0*. Alternatively, we can calculate a *p*-value.

Here we actually fit a normal distribution to the Monte carlo distribution:

pd = fitdist(zvec','Normal')

p\_fit = cdf(pd,zobs)

p\_fit = 2.1097e-04

Here we just assume that the Monte Carlo distribution is normal:

p\_theor = normcdf(zobs)

p\_theor = 1.8561e-04

Alternatively, we can calculate *p* and *z*-value directly from the sample using the Matlab function *ztest*:

[~,p,~,zval] = ztest(mc\_data,mu\_0,sigma,'tail','left')

p = 1.8561e-04

zval = -3.5597

Bootstrap

When no parametric assumptions can be made about the underlying distribution of the population that generated the random sample, we can treat the original sample as an *estimate* of the population. This estimate of the *true distribution* is called the *empirical distribution* : in this distribution each has the same likelihood of being selected in a new sample taken from .

The *bootstrap* method is based on a Monte Carlo resampling of the original sample with *replacement*. Each resampling (*bootstrap*) produces a new vector , in which some points may appear more than once or not at all. We use the notation for the *b*-th bootstrap dataset.

Typically, we will be interested in *estimators* of some population parameters , such as the mean, the variance, moments, etc.. We call these estimators *point estimates*.  It is unlikely that a point estimate obtained from a random sample will exactly equal the true value of the population parameter. If is a population parameter that we wish to estimate, and *T* denotes a statistic that we will use as a point estimate for in the *true distribution* , the observed value of the statistic in the sample (the *empirical distribution* ) is denoted as .

For each *bootstrap* dataset we can calculate a *bootstrap* replica of as .  These B bootstrap replicates provide us with an estimate of the distribution of , from which we can derive an estimate for *standard error* of and *confidence interval* for the true .

In the following example we compute the standard error for the correlation coefficient between two vectors:

load DATABASE/bootstrap\_corr

First get the sample size.

[m,n] = size(R);

Here we get the value of the statistic of interest: in this case it is the correlation between the two columns of R:

theta = corr(R(:,1),R(:,2));

Here we generate B bootstrap replicates:

B = 1000;

We use Matlab function *unidrnd* to get the indices to the resamples. Note that each column corresponds to indices for a single bootstrap resample.

inds = unidrnd(m,m,B);

Here we extract the resampling from the data using the indices.

xboot = zeros(m,n,B);

for i = 1:B

    xboot(:,:,i) = R(inds(:,i),:);

end

Here we get the value of the statistic of interest for each bootstrap replicate.

thetab = zeros(B,1);

for i = 1:B

thetab(i) = corr(xboot(:,1,i),xboot(:,2,i));

end

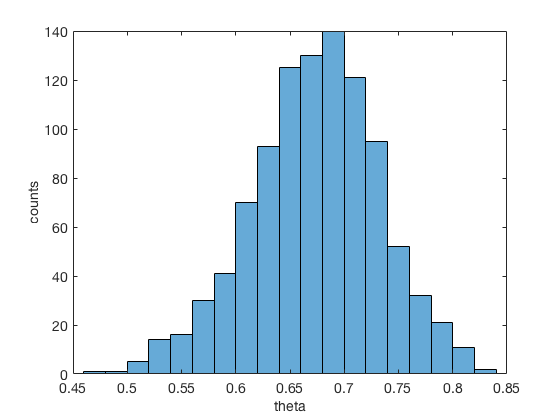
Here we plot the histogram of the statistics and calculate the standard deviation of (standard error of ):

histogram(thetab)

seb = std(thetab)

xlabel('theta');ylabel('counts')

seb = 0.0587



Alternatively, we can use the Matlab function *bootstrp* to calculate each replicate theta.

[bootstat,bootsam] = …

bootstrp(1000,@corr,R(:,1),R(:,2));

se = std(bootstat)

se = 0.0559

and *bootci* to calculate the confidence interval for :

[ci,bootstat] = …

bootci(1000,@corr,R(:,1),R(:,2));

The standard error of is one measure of the performance of the estimator. *Bias* is another measure: for a general parameter , *bias* is defined as the difference between the expected value of the *test statistic T* (used as a *point estimate* of ) and the parameter .

To get the bootstrap estimate of bias, we use the empirical distribution as before. We resample from the empirical distribution and calculate the statistic using each bootstrap resample, yielding the bootstrap replicates. Then, we use these to estimate the bias as:

The purpose of the *bias* is to correct the *estimator* , and therefore the *bias corrected estimator* is defined as:

bias = mean(bootstat) - theta

bce = 2\*theta - mean(bootstat)

se = std(bootstat)

ci

bias = 0.0028

bce = 0.6681

se = 0.0573

ci = [0.5420 0.7685]

Obviously, the key assumption underlying the theory of the bootstrap is the notion that the empirical distribution function is representative of the true population distribution . If this is not the case, then the bootstrap will not yield reliable results. For example, this can happen when the sample size is small or the sample was not gathered using appropriate random sampling techniques.

*Jackknife*

We have seen how *Bootstrap* methods choose random samples with replacement from the sample data to estimate standard error and confidence intervals for one or more parameters of interest. Sampling with replacement means that each observation is selected separately at random from the original dataset. So, a particular data point from the original data set could appear multiple times in a given bootstrap sample. Furthermore, the number of elements in each bootstrap sample equals the number of elements in the original data set.

*Jackknife* resamples systematically, rather than at random as bootstrap does. For a sample with *m* points, the jackknife computes sample statistics on *m* separate samples of size *m*-1. Each sample is the original data with a single observation omitted:

)

*Jackknife* is mostly used to estimate the *bias* and/or *standard error* of a sample statistic (see also CHAPTER 6, Special Topic: Bias vs. variance in linear regression).

We can use Matlab function *jackknife* to calculate these quantities.

jackstat = jackknife(@corr,R(:,1),R(:,2));

jbias = (m-1) \* (mean(jackstat) - theta)

bce\_j = theta - jbias

jse = sqrt((m-1)/m \* (jackstat-mean(jackstat))'\*(jackstat-mean(jackstat)))

jbias = 0.0030

bce\_j = 0.6678

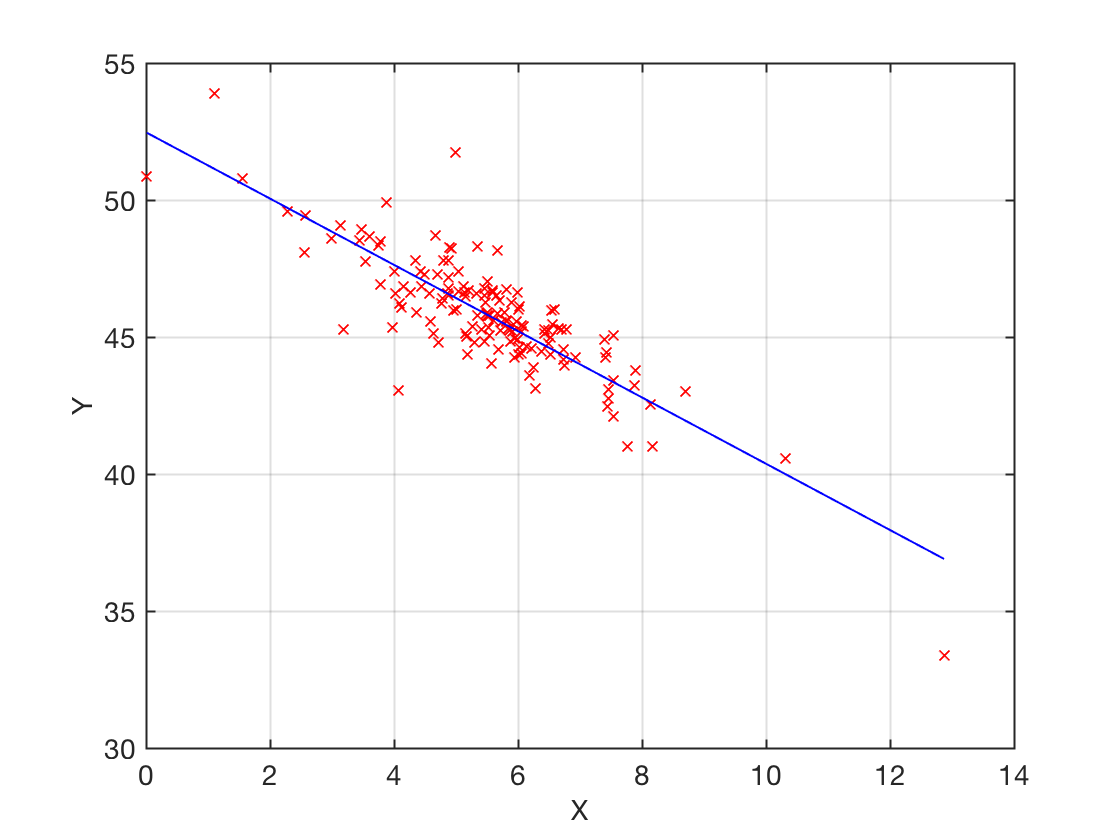
jse = 0.0634

*Cross-validation.*

Conceptually similar to *bootstrap* and *jackknife* is the method of *cross-validation*.  Very often we create a model of the relationship between variables using sample data. The goal of the model is to make predictions about the dependent variables (i.e., ***Y***) once we know the independent variables (i.e., ***X***). A common mistake is to build a model based on a data set and then to use that same data to assess the performance of the model. Cross-validation is a technique that iteratively partitions the sample into two sets of data, one that is used for building the model (the *training set*), and the other that is used to test it (the *test set*).

For example, we can use cross-validation in linear (or non-linear) regression to estimate the expected *prediction error*, PE.  The *prediction error* is the *mean squared error* (mse) of the regression in which the residual is calculated for observations that were not included in the model. For example, if the ***X*** and ***Y*** variables have 100 observations, we can split them into a *training set* of 90 () and a *test set* of *mtest* = 10 observations (). After determining the regression coefficients using only the training sets, we apply them to the to obtain a prediction, . The *prediction error*, PE, is defined as:

Most often we use*K-fold cross validation*. In this method, the data set is divided into *k* subsets. Each time, one of the *k* subsets is used as the *test set* and the other *k-1* subsets are put together to form a *training set*. Then the average *mse* across all *k* trials is computed. The advantage of this method is that it matters less how the data gets divided. Every data point gets to be in a test set exactly once, and gets to be in a training set *k-1* times. The Monte Carlo choice of the *k* subsets can be repeated several times, providing further averaging of the *prediction error*. Here is an example:



load DATABASE/crossval

Xmat = [ones(m,1) X]

beta = (Xmat'\*Xmat)\Xmat'\*Y

Y\_hat = beta(1) + beta(2)\*X;

plot(X,Y,'xr',X,Y\_hat,'b')

e = Y\_hat - Y;

sse = e'\*e;

df = m-length(beta);

mse = sse/df

mse = 1.3859

K-fold cross-validation

K = 10;

nt = m/K;

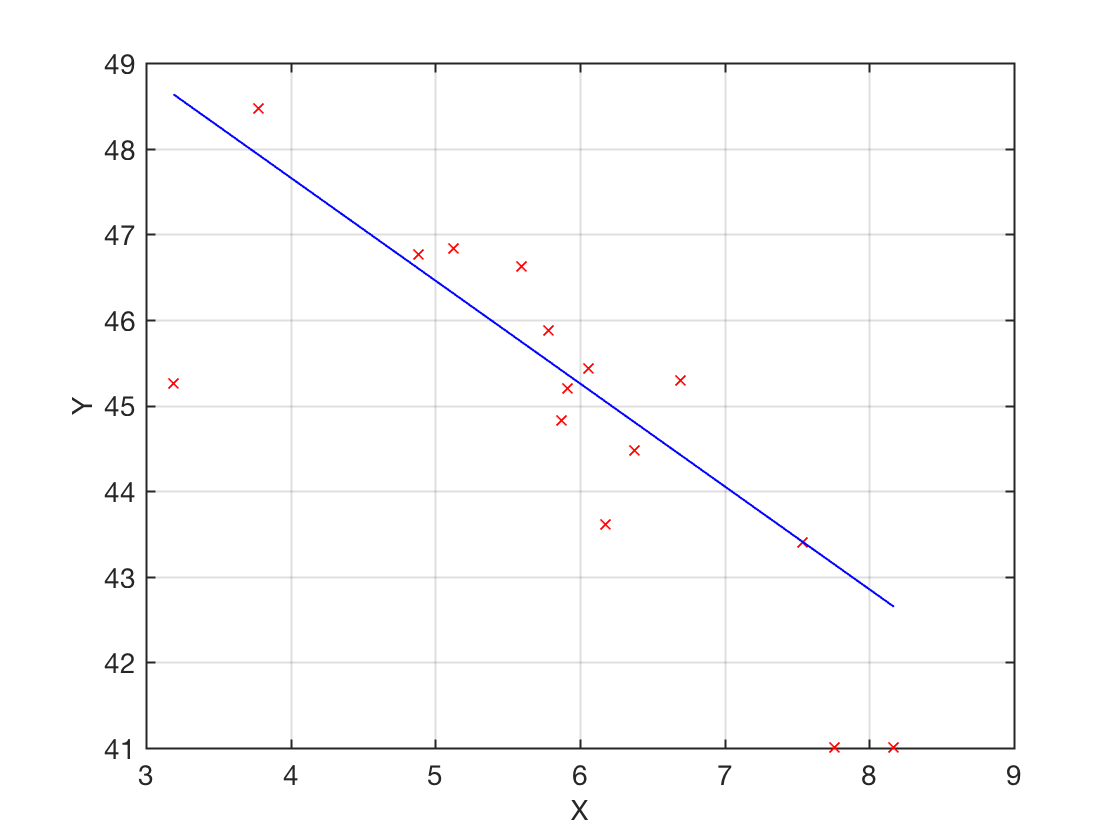
Monte Carlo repeats

nMC = 20;

pe = zeros(nMC,1);

for j = 1:nMC

sse\_test = zeros(K,1);

mse\_test = zeros(K,1);

perm\_ind = randperm(m,m);

ind = [1:nt];

check = [];

for i = 1:K

test\_ind = perm\_ind(ind);

Y\_test = Y(test\_ind);

X\_test = X(test\_ind);

Y\_train = Y;

Y\_train(test\_ind) = [];

X\_train = X;

X\_train(test\_ind) = [];

m\_train = size(X\_train,1);

Xmat\_train = [ones(m\_train,1) X\_train];

beta\_train = (Xmat\_train'\*Xmat\_train)\Xmat\_train'\*Y\_train

Y\_hat\_test = beta\_train(1) + beta\_train(2)\*X\_test;

% plot(X\_test,Y\_test,'xr',X\_test,Y\_hat\_test,'b')

e\_test = Y\_hat\_test - Y\_test;

sse\_test(i) = e\_test'\*e\_test;

m\_test = size(X\_test,1);

mse\_test(i) = sse\_test(i)/m\_test;

ind = ind + nt;

end

pe(j) = mean(mse\_test);

end

pe = mean(pe)

Alternatively we can use Matlab *crossval* function:

regf = @(XTRAIN,ytrain,XTEST)(XTEST\*regress(ytrain,XTRAIN));

cvMse = crossval('mse',Xmat,Y,'predfun',regf,'mcreps',nMC)

pe = 1.4533

cvMse = 1.4333

We can see how the *prediction error* calculated by cross-validation is higher than the *mse* calculated including all the data. A variant of this method is to randomly divide the data into a test and training set *k* different times. The advantage of doing this is that you can independently choose how large each test set is and how many trials you average over.