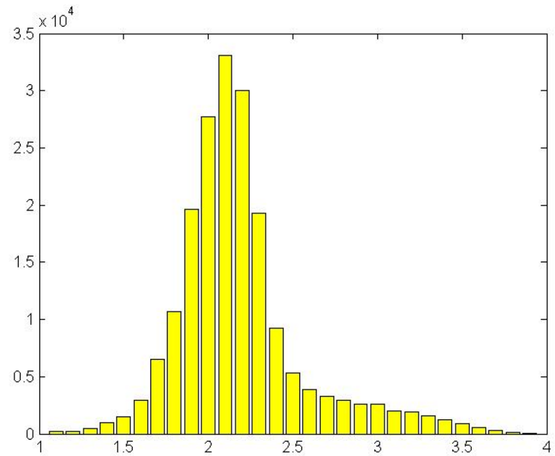
CS395T
Computational Statistics with Application to Bioinformatics

Prof. William H. Press
Spring Term, 2011
The University of Texas at Austin

Lecture 14

Let's turn from (x,y,σ) data to data that comes as counts of things.

Two common examples are "binned values" (histograms) and contingency tables.



	C_0	C_1
f_0	n_{00}	n_{01}
f_1	n_{10}	n_{11}

Counts are distributed according to (in general, unknown) probabilities p_i or p_{ij} across the bins or table entries. The model (with parameters maybe) predicts the p's.

$n_i \sim \text{Binomial}(N, p_i)$ or more precisely, $\{n_i\} \sim \text{Multinomial}(N, \{p_i\})$

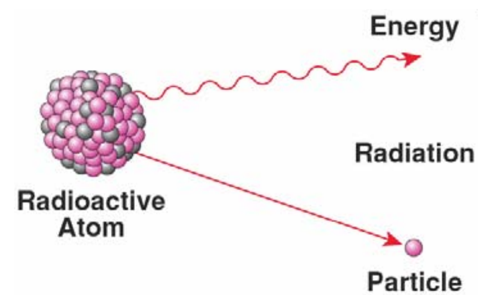
For histograms (but not necessarily contingency tables) one commonly has

$n_i \ll N \;\Rightarrow p_i \ll 1$ for all i

$n_i \ll N \;\Rightarrow p_i \ll 1$ for all i implies that counts are (close to) Poisson distributed

$$\text{Binomial}(n, N, p) \Rightarrow$$

$$\begin{aligned}
P(n) &= \frac{N!}{n!(N-n)!} p^n (1-p)^{N-n} \\
&= \frac{1}{n!} \frac{N!}{(N-n)!} p^n e^{(N-n)\ln(1-p)} \\
&\approx \frac{1}{n!} (Np)^n e^{-(Np)} \\
&\sim \text{Poisson}(Np)
\end{aligned}$$



Sometimes this is not even an approximation, but exact because of how the data is gathered. Everyone's favorite example: radioactive decays.

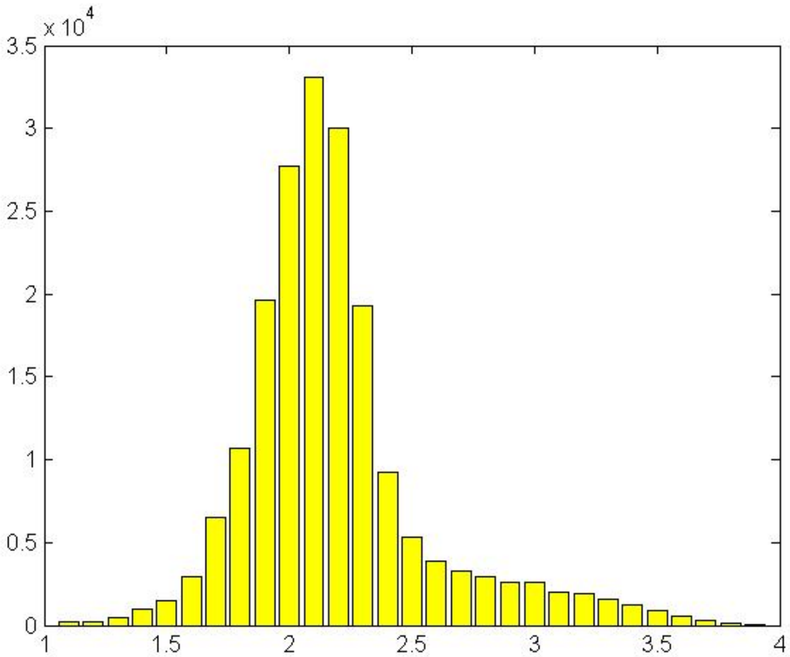
It depends on whether N was a constraint, or "just happened". We will return to this issue when we discuss contingency tables: details of the exact protocol can subtly affect the statistics of the result.

Also recall, $\quad x \sim \text{Poisson}(\lambda) \Rightarrow \mu(x) = \lambda,\; \text{Var}(x) = \lambda$

The histogram we just saw is biological:
It's the distribution of log10 of exon lengths in human.

```
exloglen = log10(cell2mat(g.exonlen));
[count cbin] = hist(exloglen,(1:.1:4));
count = count(2:end-1); % trim ends, which have overflow counts
cbin = cbin(2:end-1);
ecount = sqrt(count+1);
bar(cbin,count,'y')
```

"pseudo-count"



$$\chi^2 = \sum_i \frac{(n_i - Np_i)^2}{Np_i}$$

"Pearson"

"$(O\text{-}E)^2 / E$"

but people often use

$$\chi^2 = \sum_i \frac{(n_i - Np_i)^2}{n_i + \alpha}$$

"modified Neyman"

"$(O\text{-}E)^2 / O_m$"

"pseudo-count"

Why do they do this?

1. It's the numerator that drives the fit to the data. Denominator shouldn't matter much.
2. Many NLS algorithms/packages require σ's as input and can't fit them from a model.
3. Having the model in the denominator makes it more likely that you'll converge to a spurious local minimum (never recover from an iteration with a very small p_i).

Two asides:

1. The pseudocount can be thought of as resulting from a power-law prior on λ

```
In[1]:= poi = lam^n Exp[-lam]

Out[1]= e^-lam lam^n
```

$$P(n,\lambda) \propto \lambda^n e^{-\lambda}$$

```
In[2]:= Solve[D[poi, lam] == 0, lam]

Out[2]= {{lam → n}}
```

$$\frac{dP}{d\lambda} = 0 \Rightarrow \lambda = n$$

```
In[4]:= Solve[D[poi lam^alpha, lam] == 0, lam]

Out[4]= {{lam → alpha + n}}
```

$$\frac{d(P\lambda^\alpha)}{d\lambda} = 0 \Rightarrow \lambda = n + \alpha$$

2. We mentioned in class Matlab's lack of a weighted nonlinear fit function. We can make one out of their unweighted function nlinfit (they have a help page telling how to do this):

```
function [beta r J Covar mse] = nlinfitw(x,y,sig,model,guess)
yw = y./sig;
modelw = @(b,x) model(b,x) ./ sig;
[beta r J Covar mse] = nlinfit(x,yw,modelw,guess);
Covar = Covar ./ mse; % undo Matlab's perhaps well-intentioned scaling
```

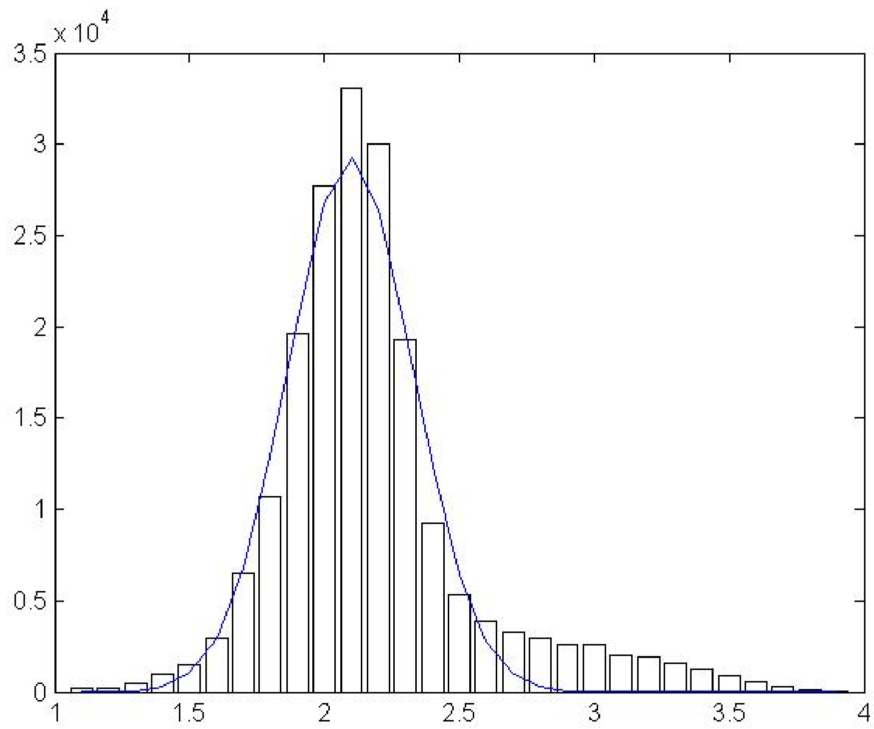
The Neyman χ^2 (previous slide) fits into this common interface to nonlinear least square (NLS), while the Pearson (truer) χ^2 doesn't.

OK, we're ready to fit a model to the exon length data.

Fit a single Gaussian (of course, it's in log space)

```
modeloneg = @(b,x) b(1).*exp(-0.5.*((x-b(2))./b(3)).^2);
guess = [3.5e4 2.1 .3];
[bfit r J Covar mse] = nlinfitw(cbin,count,ecount,modeloneg,guess);
bfit, Covar, mse
stderr = sqrt(diag(Covar))
plot(cbin,modeloneg(bfit,cbin),'b')
bfit =
       29219       2.0966      0.23196
Covar =
        8513   -0.0012396     -0.02769
  -0.0012396   3.1723e-007   9.833e-009
    -0.02769    9.833e-009  2.1986e-007
mse =
      849.37
stderr =
      92.266
   0.00056323
    0.0004689
```

"mean square error"



mse is just another name for χ^2/N, so it should be ~1 for a good fit

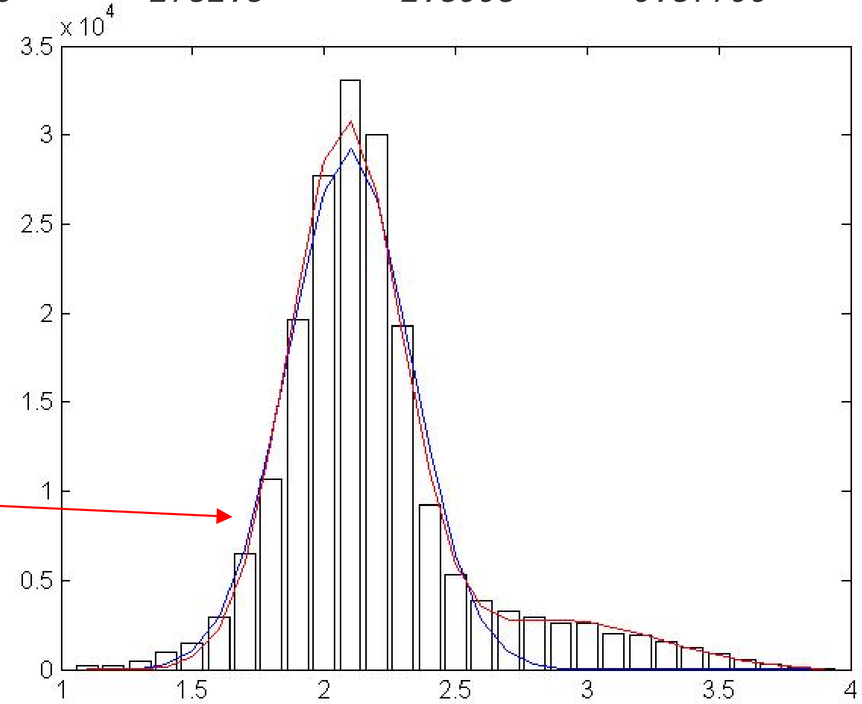
Fit sum of two Gaussians:

This time, we'll put the model function into an external file:

```
function y = modeltwog(b,x)
y = b(1).*exp(-0.5.*((x-b(2))./b(3)).^2) + ...
    b(4).*exp(-0.5.*((x-b(5))./b(6)).^2);

guess2 = [guess 3000 3. 0.5];
[bfit2 r2 J2 Covar2 mse2] = nlinfitw(cbin,count,ecount,@modeltwog,guess2);
bfit2, sqrt(diag(Covar2)), mse2
plot(cbin,modeltwog(bfit2,cbin),'r')
hold off
bfit2 =
        30633       2.0823      0.21159       2732.5       2.8998      0.37706
ans =
       99.609
   0.00069061
   0.00056174
       23.667
    0.0069429
    0.0041877
mse2 =
       163.44
```

Although it seems to capture the data qualitatively, this is still a bad fit. Whether this should worry you or not depends completely on whether you believe the model should be "exact"



We keep getting these big mse's!
Let's verify that mse ~ 1 is what you should get for a perfect model:

```
perfect = 2.0 .* randn(10000,1) + 4.;
[count cbin] = hist(perfect,(-1:.2:9));
count = count(2:end-1);
cbin = cbin(2:end-1);
ecount = sqrt(count+1);
[bfit r J Sigma mse] = nlinfitw(cbin,count,ecount,modeloneg,guess);
bfit, Sigma, mse
bfit =
       393.27        4.0159        2.0201
Sigma =
       25.784   -0.0005501     -0.057248
   -0.0005501   0.00046937   3.5555e-006
    -0.057248  3.5555e-006    0.00032997
mse =
      0.95507
chisq = numel(count)*mse
df = numel(count)-3;
pvalue = chi2cdf(chisq,df)
chisq =
       46.799
pvalue =
      0.56051
```

Let's see if 0.955 is actually good enough:

by definition, mse times number of bins equals chi-square

three fitted parameters

yep, good enough!

If you expect the model to be exact, then take the p-value seriously.

If you don't, it is called "chi-by-eye" (term used as an insult in the physical sciences).

It's OK when the intent of the model is to summarize main features of the data without necessarily fitting it exactly.

The Poisson-count pitfall: $\chi^2 = \sum_i \frac{(x_i - \mu_i)^2}{\mu_i}$ is actually not *Chisquare* !

You can get a statistic that is "accurately" chi-square either by summing (any number of) terms that are accurately squares of Normal t-values,

or by summing a large number of terms that individually have the correct mean and variance. This uses the CLT, so the exactness of chi-square is no better than its normal approximation.

Compute moments of chi-square with 1 d.f.:

```
In[31]:= py = (1 / (Sqrt[2 Pi y])) Exp[-(1 / 2) y]
```

Out[31]=

$$\frac{e^{-y/2}}{\sqrt{2\pi}\sqrt{y}}$$

```
In[32]:= Integrate[py {1, y, y^2}, {y, 0, Infinity}]
```

Out[32]=

```
{1, 1, 3}
```

So, $\mu = 1, \quad \sigma^2 = 3 - 1 = 2$

Hence, $\text{Chisquare}(\nu) \to \text{Normal}(\nu, \sqrt{2\nu})$ as $\nu \to \infty$

If you are going to rely on the CLT and sum up lots of not-exactly-t bins, they must have the expected mean and variance.

Poisson doesn't have. (People often get this wrong!)

```
In[39]:= poi[n_] := Exp[-mu] mu^n / n!

In[48]:= poimean = Sum[n poi[n], {n, 0, Infinity}]

Out[48]=
        mu
```

OK

```
In[50]:= poivar =
          Simplify[Sum[n^2 poi[n], {n, 0, Infinity}] -
            poimean^2]

Out[50]=
        mu
```

OK

```
In[51]:= tmean = Sum[((n - mu)^2 / mu) poi[n], {n, 0, Infinity}]

Out[51]=
        1
```

OK

```
        tvar =
         Simplify[
          Sum[((n - mu)^2 / mu)^2 poi[n], {n, 0, Infinity}] -
           tmean^2]

Out[53]=
```

$$2 + \frac{1}{mu}$$

Not OK!

Poisson, Pearson chi-square statistic: $\chi^2 = \sum_i \frac{(x_i - \mu_i)^2}{\mu_i}$

We now know that this χ^2 is not Chi-square distributed! Rather, asymptotically,

$$\chi^2 \sim \text{Normal}\left(\nu, 2\nu + \sum_i \mu_i^{-1}\right)$$

What about bins with μ near zero?
(Decide in advance!)

I wonder if Modified Neymann, $\chi^2 = \sum_i \frac{(n_i - Np_i)^2}{n_i + \alpha}$ is any closer to true chisquare?

```
poi = @(n,mu) exp(-mu).*mu.^n./factorial(n);
mus = [0.1 0.5 1.0 1.5 2.0 3 5 7 10 20 30];
nsum = 200;
for j=1:numel(mus),
   mu = mus(j);
   pois = poi(0:nsum,mu);
   ts = ((0:nsum)-mu).^2 ./ mu;
   tas = ((0:nsum)-mu).^2 ./ ((0:nsum)+1);
   tmean = sum(ts.*pois);
   tamean = sum(tas.*pois);
   tvar = sum(ts.^2.*pois)-tmean^2;
   tavar = sum(tas.^2.*pois)-tamean^2;
   fprintf(1,'%4.1f %8.5f %8.5f %8.5f
      %8.5f\n',mu,tmean,tamean,tvar,tavar);
end
```

0.1	1.00000	0.05147	12.00000	0.01954
0.5	1.00000	0.27061	4.00000	0.05447
1.0	1.00000	0.52848	3.00000	0.25011
1.5	1.00000	0.73696	2.66667	0.80999
2.0	1.00000	0.89099	2.50000	1.70583
3.0	1.00000	1.06780	2.33333	3.85907
5.0	1.00000	1.15149	2.20000	6.40207
7.0	1.00000	1.13452	2.14286	6.24527
10.0	1.00000	1.09945	2.10000	4.88251
20.0	1.00000	1.05000	2.05000	3.10583
30.0	1.00000	1.03333	2.03333	2.68296

Wow, it’s much worse! I never knew that!
Verdict: Don’t use Modified Neymann for a goodness-of-fit test unless the number of counts is way large!

Remember our three ways of computing the uncertainty in other quantities?

For example, what if we want the ratio of areas in the two Gaussians?

Method 1: Linearized propagation of errors

Recall the meaning of the b's:

```
function y = modeltwog(b,x)
y = b(1).*exp(-0.5.*((x-b(2))./b(3)).^2) + ...
    b(4).*exp(-0.5.*((x-b(5))./b(6)).^2);
```

We start off in Mathematica:

```
b = {b1, b2, b3, b4, b5, b6};

func = (b1 * b3) / (b4 * b6)
```

$$\frac{b1\,b3}{b4\,b6}$$

```
symgrad = D[func, {b}]
```

$$\left\{\frac{b3}{b4\,b6},\ 0,\ \frac{b1}{b4\,b6},\ -\frac{b1\,b3}{b4^2\,b6},\ 0,\ -\frac{b1\,b3}{b4\,b6^2}\right\}$$

```
bfit2 = {30633, 2.0823, 0.21159, 2732.5, 2.8998, 0.37706};

grad = symgrad /. ToRules[b == bfit2]
```

Mathematicaology!

```
{0.000205364, 0, 29.7316, -0.00230226, 0, -16.6841}
```

And then switch to MATLAB, yikes!

```
mu = bfit(1)*bfit(3)./(bfit(4)*bfit(6))
sigma = sqrt(grad' * Covar2 * grad)
mu =
       6.2911
sigma =
     0.096158
```

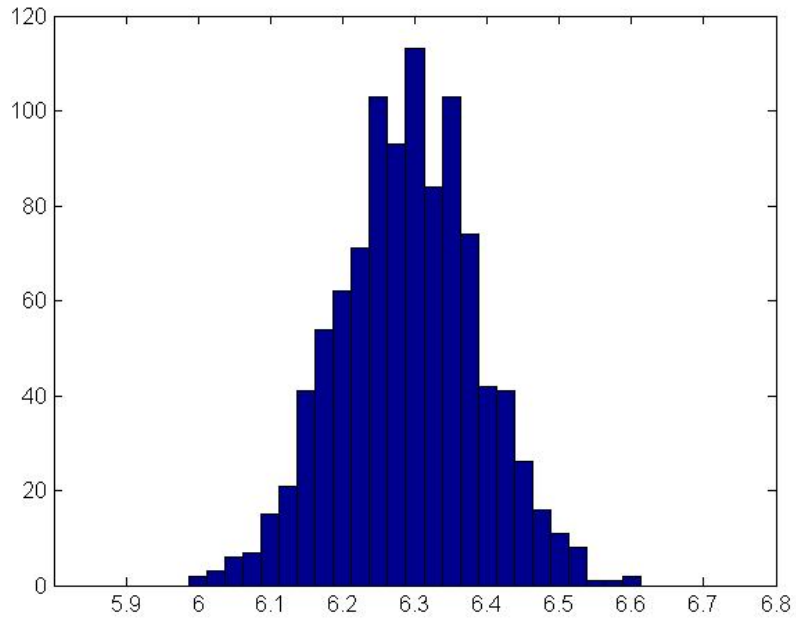
6.2911: ratio of the areas

0.096158: its standard error

Method 2: Sample from the posterior distribution

```
samp = mvnrnd(bfit2,Covar2,1000);
samp(1:5,:)
ans =
        30430         2.082       0.21203        2754.6        2.8975       0.37636
        30421        2.0829       0.21213        2701.1        2.9026       0.37738
        30645        2.0815       0.21125        2775.3        2.8969       0.37969
        30548        2.0822       0.21229        2714.7        2.9011       0.37712
        30607        2.0826       0.21175          2718        2.9016       0.37779
funcsam = (samp(:,1).*samp(:,3))./(samp(:,4).*samp(:,6));
funcsam(1:5)
ans =
       6.2234
       6.3307
       6.1437
       6.3346
       6.3116
hist(funcsam,[5.9:0.025:6.7]);
mu = mean(funcsam)
sigma = std(funcsam)
mu =
       6.2911
sigma =
     0.096832
```

multivariate Normal random generator



Method 3: Bootstrap

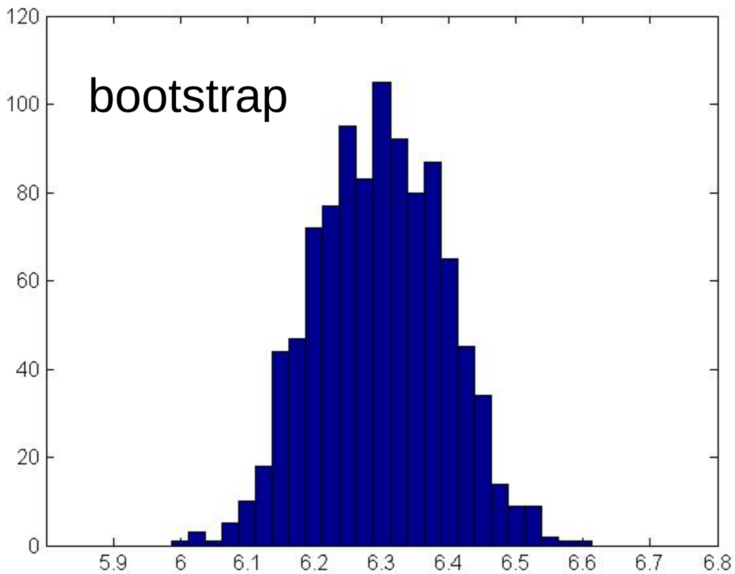
```
function mu = areasboot(data)
samp = randsample(data,numel(data),true);
[count cbin] = hist(samp,(1:.1:4));
count = count(2:end-1);
cbin = cbin(2:end-1);
ecount = sqrt(count+1);
guess = [3.5e4 2.1 .3 3000 3. 0.5];
[bfit r J Covar mse] = nlinfitw(cbin,count,ecount,@modeltwog,guess);
mu = (bfit(1)*bfit(3))/(bfit(4)*bfit(6));
```

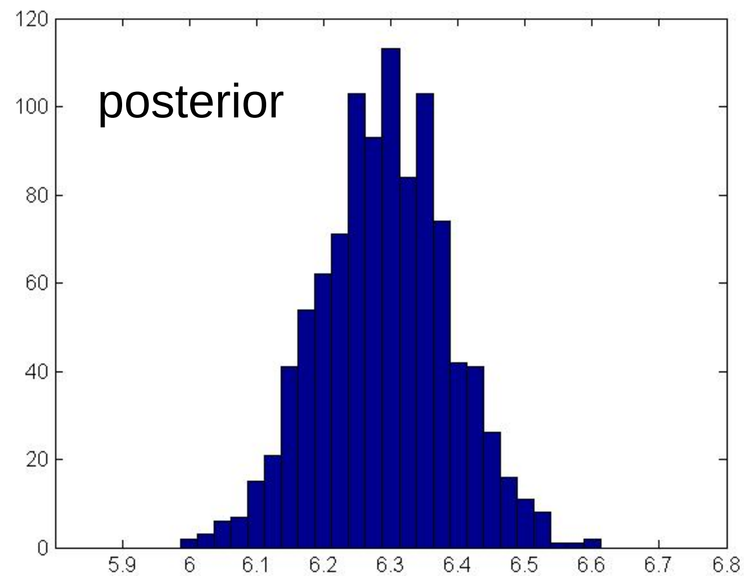
list of individual exon lengths. Notice that we resample before (re-)binning.

```
areas = arrayfun(@(x) areasboot(exloglen), (1:1000));
mean(areas)
std(areas)
ans =
      6.2974
ans =
    0.095206
hist(areas,[5.9:0.025:6.7]);
```

takes about 1 min on my machine

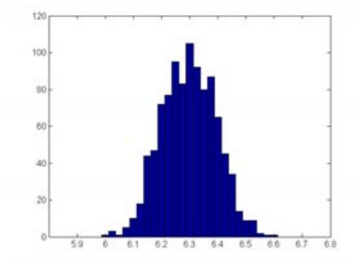
recall, sampling from the posterior gave:

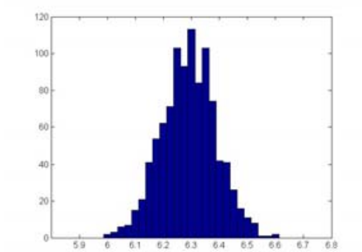


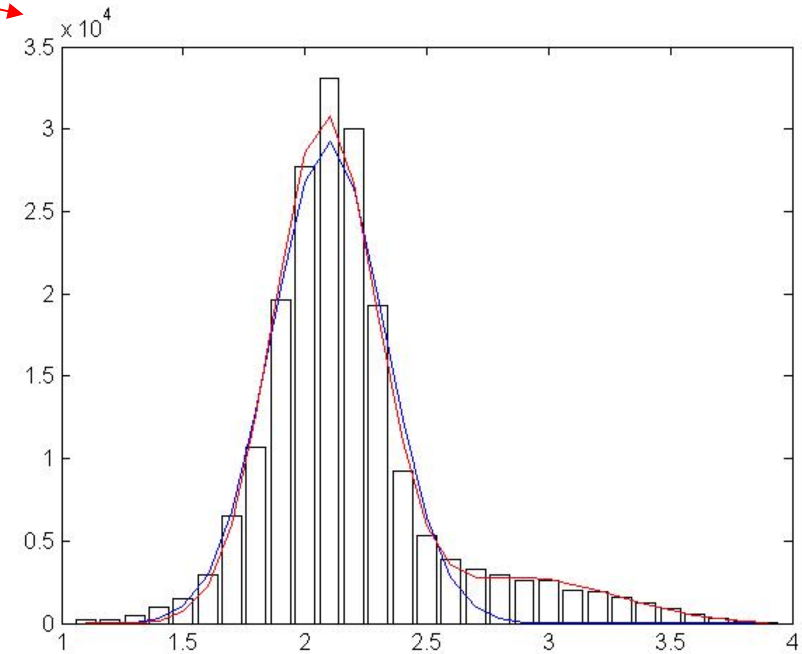


Everything works out nicely here because we have lots of data.
We are in “asymptopia”!

Ratio of areas $= 6.3 \pm 0.1$







But remember that we did “chi by eye” here. (The model is not a perfect fit.)

Our value and uncertainty “are what they are” within the imperfect model.
They have no magical power to peer into the underlying heart of nature!

We’ll come back to this data set later.