Chapter 4. The BOOTSTRAP

4.1. Normal Errors

The definition of error of a fitted variable from the variance-covariance method relies on one assumption- that the source of the error is such that the "noise" measured has a normal distribution. In case you don't remember what a normal distribution is, it's a bell-shaped curve (or Gaussian). In case you are not sure what it means for noise to have a normal distribution, take the following set of data:

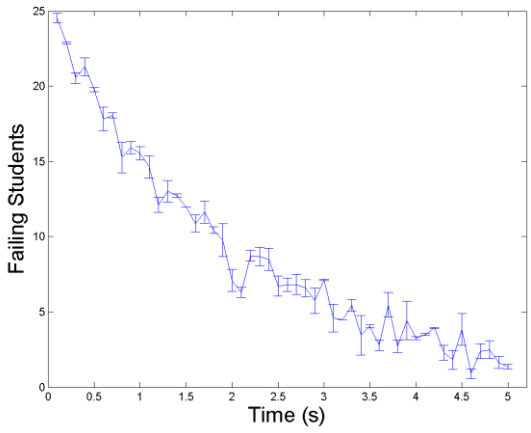

Figure 1

Now, I make a histogram of the magnitude of the errorbars:

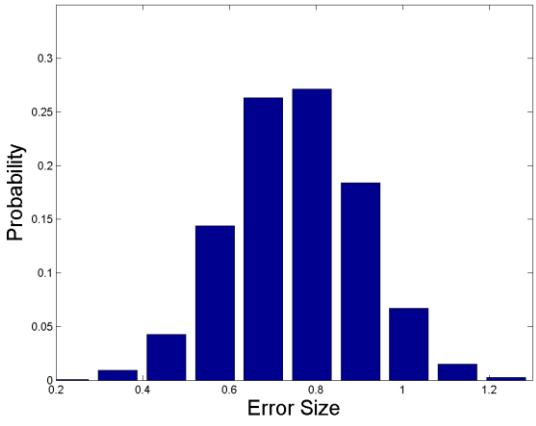

Figure 2

See- you can tell the error distribution is bell-shaped! That means that there is a decreasing probability to get a point with an error that is very far away from the average error, and there is no error bar as small as 0 (i.e. there is always a little bit of error). Generally, this makes a lot of sense as our word for "outliers" implies something that is away from the norm.

So when you're fitting data with normally distributed errors, you can analyze the errors to the fits with the variance-covariance matrix method which I am still waiting for most of you to do.

4.2 Abnormal Errors

Now let's look at this exponential decay curve:

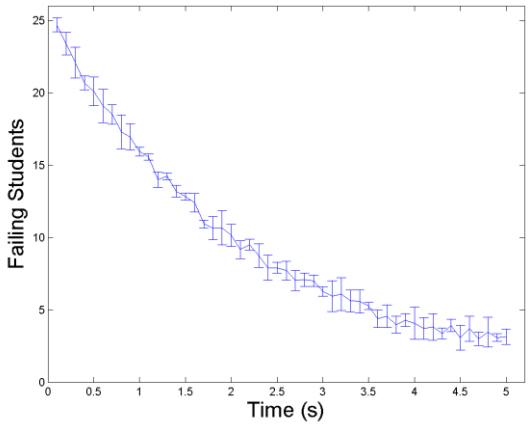

Figure 3

Here is a histogram of the errorbars:

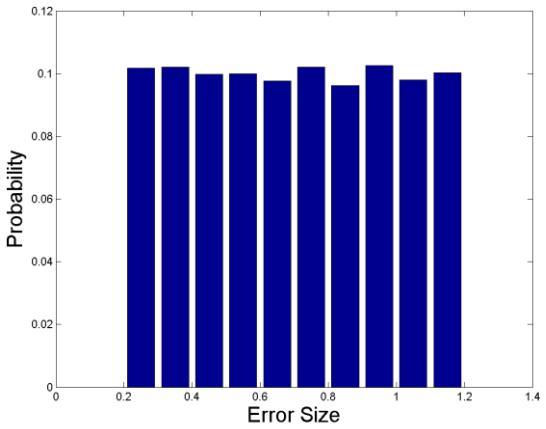

Figure 4

While the magnitude of the error bars span a similar range as before (0.2→1.2 failing students), this does ***not*** look like a bell-shaped curve but a flat line. In this case, the variance-covariance method will **FAIL**.

The solution is the Bootstrap Method. In this procedure, you will first fit your data as best you possibly can by minimizing χ^2. This gives you the best fit but you don't know how wide the variances to the parameters of that fit are. In the next step, you take that best fit, add noise that conforms to your distribution of noise, and you re-fit this new set of data. This gives you an all-new set of parameters to your best fit. Now you repeat this process several times giving you a very large dataset of parameters to work with. The variance in the parameters can then be estimated by the standard deviation of all those fitted parameters.

Step 1. Fit the data by minimizing χ^2. In this example case, this is an exponential decay: I calculate an amplitude of 25.3581 failing students and a rate constant of 0.4579 s^{-1} (or τ = 2.1839 s) using a script I call fitter4; the data is stored in three columns in the file 'Dataset4_1.txt':

```
function [return_val]=fitter4(x)
err=0;
mydata4=load('Dataset4_1.txt');
for i=1:50
 err=err+(1/mydata4(i,3)^2)*(mydata4(i,2)-x(1)*exp(-mydata4(i,1)*x(2)))^2;
end;
return_val=err;
```

In the consul, I typed:

```
>> mydata4=load('Dataset4_1.txt');
>> x(1)=25; x(2)=1;
>> fminsearch('fitter4',x)
ans =
  25.3581    0.4579
>> x2=ans;
```

In case you forgot, the point of the above is to find the parameters for the fit that minimize χ^2, which is:

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{1}{\sigma(i)^2}\right) [\text{data}(i) - \text{fit}(i)]^2$$

Its fairly frequent to see the effect of the individual errors associated with each point to not be included, in which case $\chi^2 = \sum_{i=1}^{N} [\text{data}(i) - \text{fit}(i)]^2$. Most analytical instruments report data without the associated σ, which is why this is done.

Step 2. Make a fit from these best parameters:

```
>> for i=1:50 fit(i)=x2(1)*exp(-mydata4(i,1)*x2(2)); end;
```

Here is a plot; the data are in blue, the fit in red; it looks like a very good fit.

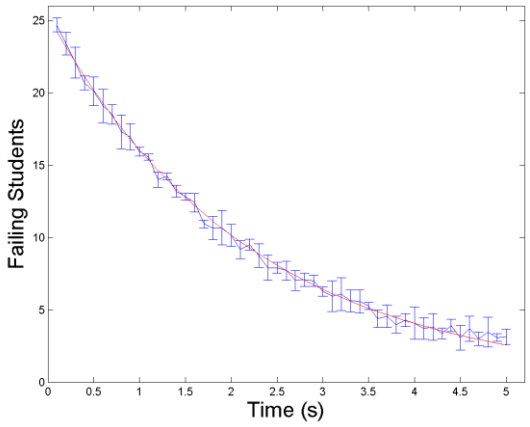

Figure 5

Step 3. Add noise to this fit from the known distribution of error and treat this as a new set of data. This is called the **Monte-Carlo** method. Re-calculate the best fit parameters to this "fake" data.

Now this is where some of you might get flustered, but it is actually quite simple. In the example above, the error is centered at ~0.7 and varies fairly evenly from 0.2 → 1.2. This means that the data point can either be in error no less than 0.2 failing students or no more than 1.2 failing students, and that it is equally probably that the error is somewhere within this range. Now I know that Matlab can make evenly (i.e. flat) distributed random numbers from 0→1 using the **rand** command.
To explore this, do the following:

>> for i=1:10000 catinhat(i)=rand; end;	This is just a series of 10000 random numbers.
>> hist(catinhat);	This is how you make histograms.
>> h_legend=xlabel('Magnitude')	These commands allow you to make the
>> set(h_legend,'FontSize',20);	x-axis and y-axis labels bigger. I'm just
>> h_legend=ylabel('Number of occurrences');	showing you how for the heck of it.
>> set(h_legend,'FontSize',20);	
>> axis([-.5 1.5 0 1200]);	

To aid your understanding, just cut and paste the line below into matlab (it's the same commands):

```
for i=1:10000
catinhat(i)=rand;
end; hist(catinhat);
h_legend=xlabel('Error Size');
set(h_legend,'FontSize',20);
h_legend=ylabel('Number of occurrences');
set(h_legend,'FontSize',20);
axis([-.5 1.5 0 1200]);
```

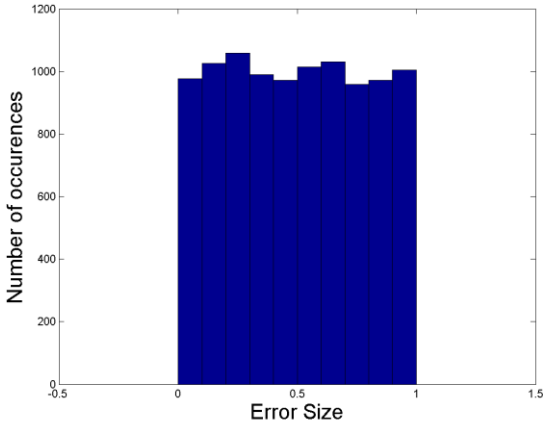

Figure 6

Remember that histograms are like probability distributions; in this case I see that it generates random numbers evenly from 0 → 1. To make random numbers from 0.2 → 1.2, I just add 0.2 from rand:

```
>>for i=1:10000 catinhat(i)=rand+0.2; end;
>>hist(catinhat);
```

To save time, cut and paste the following into matlab:

```
for i=1:10000
catinhat(i)=rand+.2;
end;
hist(catinhat);
h_legend=xlabel('Error Size');
set(h_legend,'FontSize',20);
h_legend=ylabel('Number of occurrences');
set(h_legend,'FontSize',20);
axis([0 1.4 0 1200])
```

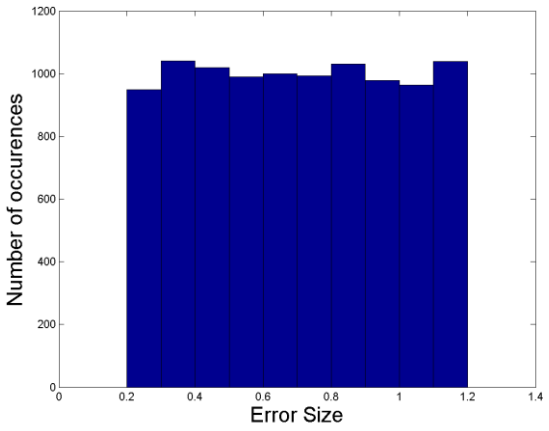

Figure 7

Note that the random numbers spread from 0.2→ 1.2, which is just like the error distribution in the data that I am analyzing; it's very similar to Fig. 4. We are almost done.

To make our first Monte-Carlo set of data, we now take the fit and add the "flat" random numbers from 0.2 to 1.2. **NOTE! In reality, errors can be positive or negative!** To account for this, we multiply the magnitude of the error by sign(randn), a function that randomly makes the error positive or negative:

```
>> for i=1:50 montefit(i)=fit(i)+(rand+0.2)*sign(randn); end;
```
This adds noise to the fit

```
>> plot(mydata4(:,1),montefit);
```
Plots the "fake data"

```
>> hold on;
>> plot(mydata4(:,1),mydata4(:,2),'r');
```
Overlays "fake" and real data

Here is the same for you to cut and paste into Matlab:

```
for i=1:50
montefit(i)=fit(i)+(rand+0.2)*sign(randn);
end;
plot(mydata4(:,1),montefit);
hold on;
plot(mydata4(:,1),mydata4(:,2),'r');
h_legend=xlabel('Time (s)');
set(h_legend,'FontSize',20);
h_legend=ylabel('Failing Students');
```

set(h_legend,'FontSize',20);

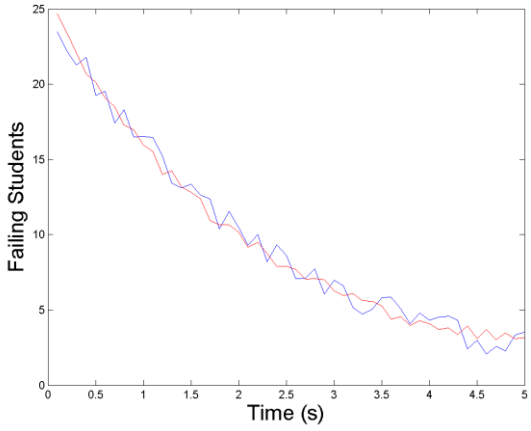

Figure 8

Notice that you can't really tell which is the "real" set of data, and which is your Monte-Carlo data!

Now when I fit this set of Monte-Carlo data, I first saved it as follows:

>> save -ascii montefit.txt montefit;

And I rewrote my fitting function and saved it as fitter4_2.m:

```
function [return_val]=fitter4_2(x)
err=0;
mydata4=load('Dataset4_1.txt');
montefit=load('montefit.txt');
for i=1:50
   err=err+(1/mydata4(i,3)^2)*(montefit(i)-x(1)*exp(-mydata4(i,1)*x(2)))^2;
end;
return_val=err;
```

Now run it:

>> fminsearch('fitter4_2',x2)
ans =
  25.7957 0.4662

Remember that x2 is your previous best fit parameters to the real set of data you fit earlier

I get A= 25.7957 failing students and rate k=0.4662 s^{-1}.
Now if I do this one more time:

>> for i=1:50 montefit(i)=fit(i)+(rand+0.2)*sign(randn); end;
>> save -ascii montefit.txt montefit;
>> fminsearch('fitter2',x2)
ans =
  25.6746 0.4620

I get A=25.6746 failing students and rate of k=0.4620 s^{-1}. Now if I generate another set of "fake" data and refit, I get A=25.6226 failing students and rate of k=0.4629 s^{-1}. Now you see I have a statistically significant set of data; I can calculate the error of the fitted amplitude by the following:

```
>> amps=[25.7957, 25.6746, 25.6226];
>> std(amps)
ans =
   0.0888
```

The result is that the best fit to the amplitude of the data is A=25.3581 ± 0.0888 failing students, or if you **would actually like to properly report the result**, its A=25.36 ± 0.09 failing students. The same analysis of the three Montecarlo rates yields k=0.4579 ± 0.0022 s^{-1} (or k=0.458 ± 0.002 s^{-1}). **Note that I am reporting the value from the fit to the "real" data, whereas I get the errors from the standard deviation from the "fake" data.** Also note that you do not report the standard deviation of the mean, just the standard deviation.

Here is an interpretation of what just happened- given that your error is random, you were just as likely to have taken the data that you "simulated" with the Monte-Carlo method as the data that you actually took (see Fig. 8). Thus, the Monte-Carlo fits you just made are as valid as the fit to the real data. In that case, the standard deviation of those fits is a meaningful statistical description of the errors of your fits.

Now while the above example works just fine, you should realize that you probably should make more than just three sets of Monte-Carlo data. The point of DAS BOOTSTRAP is that you can make hundreds or thousands (I typically make ~10,000 to 1 Billion) of such simulations. Thus, while you're fitting fake data which is normally kinda bad, you can fit so much fake data that the results are actually very meaningful. In our example above, let's make 100 Monte-Carlo fits- here is a set of examples so that it isn't so hard. Start with making the Monte-Carlo data:

```
>>for j=1:100 for i=1:50 montefit2(i,j)= fit(i)+(rand+0.2)*sign(randn); end; end;
```

You can actually plot several of them (let's do the first 3) at once to see what is happening:

```
>> plot(mydata4(:,1),montefit2(:,1:3))
```

Now you already have a function that calculates χ^2, we are just going to write a series of commands to get it to do that over and over again:

```
for j=1:100
montefit=montefit2(:,j);
save -ascii montefit.txt montefit;
params(:,j)=fminsearch('fitter4_2',x);
end;
```

(hint, write the whole thing to a script so that you can edit it more easily for your own homework: It's no different than writing the exact same thing in the consul, save it as filename.m and type >>filename <enter> in the consul).

Now you might be waiting for a few minutes…

DONE!

See, this might be the simplest exercise to date. Now the errors in my amplitude are calculated by taking the standard deviation of all the amplitude fits that are in the first row of the variable params:

```
>> std(params(1,:))
ans =
   0.5391
```

which gives an amplitude of A=25.4 ± 0.5 failing students; likewise, the rate is k=0.458 ± 0.012 s^{-1}. Note that the result is somewhat different when you ran just three Monte-Carlo simulations; this shows you that you need to perform a lot of these simulations. To check, I performed 1000 simulations, and I get the same results as when I ran just 100 simulations.

Matlab Assignment

1. The Bootstrap. The file "Problem4.txt" is part of this packet. It contains 3 by 500 data points: the first column represents time in seconds, the second Turkey Giblets, and the third is the σ of Turkey Giblets. It is roughly exponential, which is a function of the form:

$$f(t) = A \cdot e^{-k \cdot t}$$

where A is the amplitude and k is the rate or decay constant.

First, use fminsearch to calculate the best fit amplitude and decay constant. In this case, be sure to weigh the contribution to χ^2 by σ^2 of each data point. Don't forget you have to make a guess at the amplitude (it's about 100 Turkey Giblets) and decay constant (it's about 0.01 s^{-1})! Next, make a fit using the best amplitude and decay constant you calculate from fminsearch.

Now you have to use the fit to create fake data. First, histogram the σ of the data and determine what the distribution of the σ is, i.e.

```
>>hist (Problem4(:,3)); <enter>
```

I will only give it to you that it is a "flat" distribution like the example I provided in class and in the handout. Note that it isn't exactly like the example in class…

Next, create 1000 random sets of data and analyze each one with fminsearch. **Then report to me the amplitude and decay constants from the best fit to the real data, as well as the errors of the same as determined by the bootstrap method** (that's the standard deviation of the 1000 fits to the fake data, see my example handout).

-Note you **don't report the average amplitude or decay rate from the "fake data"** although those should be very close to the amplitude and decay rate you calculate from minimizing χ^2 of the real data using fminsearch. This is a good way to check that your programs are working properly.

-Note that as you are using a random number generator in this process, **absolutely none of your errors will be exactly the same** as anyone else's.

Answer:

A: 99.0392 ± 2.4079 k: 0.0101 ± 5.8837e-04

Note that your result will vary due to the Monte Carlo nature of the analysis.