Basic Statistics pt1: Fitting

Carver J. Bierson

February 2, 2018

Caveat: In this class I will give you some simple statistical tools and teach you how to use them. This is not a statistics course so we will not go into much detail about when different measures are meaningful. Different statistical techniques and measure often have different assumptions built into them that we will only touch on. When applying these tools on real data make sure that you understand your statistical tools and they apply to your data.

Later in the course we will revisit many of these topics to discuss goodness of fit and choosing the correct model for your data.

1 Random numbers

When doing statistics and testing statistics functions it is helpful to be able to generate random data. For this class we will use the numpy.random module. This module contains functions to draw values from many different distributions. Below is a basic example of generating some random data.

```
import matplotlib.pyplot as plt

import numpy.random as rnd

set_uni=rnd.random(int(1E3)) # generate 1000 random uniformly
    distributed values

set_norm=rnd.randn(int(1E3)) # generate 1000 random normally
    distributed values


# Plot data
plt.figure()

Nbins=50 # number of bins in histogram

plt.subplot(121) # subplot for uniform data
plt.title('Uniform')
plt.hist(set_uni, Nbins, facecolor='green')

plt.subplot(122)
plt.title('Normal')
plt.hist(set_norm, Nbins, facecolor='red')

plt.show()
```

Listing 1: Create and plot a histogram of random points

1.1 Seeds

Computers can't generate truly random numbers. Instead they use sophisticated mathematical formulas that simulate random values. As an aside generating truly random values is a large area of research in computer science with applications in casinos and security. Most random number generation relies on a *seed* value. If this seed is the same, you will always get the same values. If you include

```
rnd.seed(1)
```

in the code above (before calling random) it will always produce the same 'random' numbers. Python seems to be good and using a changing quantity to supply this seed so you shouldn't need to worry about this too much (If you just run the same code twice you get different values). If you do find yourself in a case where you need a new seed using the least significant figure of the time is a common method.

2 Descriptive Statistics

Numpy gives us easy to use functions to calculate many of basic statistical properties of a data set. In addition to mean, median, and standard deviation numpy will also calculate data value quantiles and percentiles. A percentile gives the data value where X percent of the data is less than that value. For example the 25th percentile is the value that is larger than 25% of the data and the 50th percentile is the median. Percentiles are more robust to outliers than the standard deviation.

Below I use a random data set as an example.

```
import numpy as np
import numpy.random as rnd

set_norm=rnd.randn(int(1E3)) # generate 1000 random normally distributed values

print('The Mean is {:0.2f}'.format(np.mean(set_norm))) # Print the data mean

print('The median is {:0.2f}'.format(np.median(set_norm))) # Print the data median

print('The standard deviation is {:0.2f}'.format(np.std(set_norm))) # Print the data standard deviation

print('The maximum value is {:0.2f}'.format(np.max(set_norm))) # Print the maximum value


print('The 75th and 99th data percentile are {:0.2f}, {:0.2f}'.format(
        np.percentile(set_norm,75),
        np.percentile(set_norm,99))) # Print the data mean
```

Listing 2: Create and plot a histogram of random points

3 Fitting

Often with data we are interested in fitting some sort of model to that data and evaluating the quality of fit. Almost all model fitting is based on minimizing the squared error,

$$E = \sum_i (p(x_i) - y_i)^2 \tag{1}$$

where y_i is one of our data points and $p(x_i)$ is some model evaluated at the x value that corresponds to y_i. I will also show some basic error estimates for fits. Note that often the error estimates that are return are under-estimates of the real error.

4 Linear fits

Lets start with the simple example of trying to fit a line.

```
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as rnd

# Generate some fake data
Datax=rnd.random(50)*10 # start with 50 uniform points

# Set the 'true' solution to y=3*x-2
Datay= 3*Datax-2

# add noise with a standard deviation of 2 to the data
Datay+=2*rnd.randn(50)


# find best linear fit
fit=np.polyfit(Datax,Datay,1) # 1 is the power of the fit, 3 would be a cubic fit

print(fit)

Plotx=np.linspace(0,10) # create an x array for plotting the fit
Ploty=np.polyval(fit,Plotx)

plt.figure()
plt.plot(Datax,Datay,'b.',label='Data') #plot data as points
plt.plot(Plotx,Ploty,'r-', label='Fit') #plot data as points

plt.legend()
```

In the example above I start with the equation $y = 3x - 2$. When I ran the code I got a fit of $y = 3.1x - 2.6$. We would like to put error estimates on those fit values to evaluate if we are really doing a good job. For this we simply need to return more values from polyfit. If we return the covariance matrix from polyfit, the diagonal elements of that matrix give the estimated standard deviation of each paramter. cov matrix contains σ^2, not σ. Should be corrected

```
import numpy as np
import numpy.random as rnd

# Generate some fake data
Datax=rnd.random(50)*10 # start with 50 uniform points

```

```
# Set the 'true' solution to y=3*x-2
Datay= 3*Datax-2

# add noise with a standard deviation of 2 to the data
Datay+=2*rnd.randn(50)


# find best linear fit
fit,covariance=np.polyfit(Datax,Datay,1,cov=True) # return
    covariance matrix

print('Fitting model y=mx+b')
print('m={:0.2f}+/-{:0.2f}'.format(fit[0],2*covariance[0,0]))
print('b={:0.2f}+/-{:0.2f}'.format(fit[1],2*covariance[1,1]))
```

When I ran this code I got. Your results will vary a bit each time because of the random arrays

```
Fitting model y=mx+b
m=3.00+/-0.02
b=-2.39+/-0.61
```

4.1 Log space fitting

Many equations in science take the form

$$y = Ae^{bx} \tag{2}$$

or

$$y = Ax^{b} \tag{3}$$

Lets say we want to solve for the parameters A and b. We can solve both of these types of equations through a simple linearization. For both equation I will take the log of both sides and use some log algebra.

$$\begin{aligned} y &= Ae^{bx} && (4)\\ \ln(y) &= \ln(Ae^{bx}) && (5)\\ \ln(y) &= \ln(A) + bx && (6)\\ & && (7) \end{aligned}$$

and

$$\begin{aligned} y &= Ax^{b} && (8)\\ \ln(y) &= \ln(Ax^{b}) && (9)\\ \ln(y) &= \ln(A) + b\ln(x) && (10) \end{aligned}$$

Below is a code solving the second form.

```
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as rnd

# Generate some fake data
Datax=1+rnd.random(50)*10 # start with 50 uniform points

```

```
# Set the 'true' solution to y=3*x^2.5
Datay= 3*Datax**2.5

#move to logspace
lnDatay=np.log(Datay)
lnDatax=np.log(Datax)

# add noise in the logspace with a standard deviation of 0.5 to the data
lnDatay+=0.5*rnd.randn(50)


# find best linear fit
fit,covariance=np.polyfit(lnDatax,lnDatay,1,cov=True) # return covariance matrix

A=np.exp(fit[1])
b=fit[0]
Astd=np.exp(covariance[1,1])
bstd=covariance[0,0]

print('Fitting model y=Ax^b')
print('A={:0.2f}+/-{:0.2f}'.format(A,2*Astd))
print('b={:0.2f}+/-{:0.2f}'.format(b,2*bstd))


Plotx=np.linspace(1,11) # create an x array for plotting the fit
Ploty=np.exp(np.polyval(fit,np.log(Plotx)))

# Plot the data in log space
plt.figure()
plt.plot(lnDatax,lnDatay,'b.',label='Data') #plot data as points in log space

plt.xlabel('ln(x)')
plt.ylabel('ln(y)')

plt.legend()

# plot in linear space
plt.figure()
plt.plot(Datax,Datay,'b.',label='Data') #plot data as points
plt.plot(Plotx,Ploty,'r-', label='Fit') #plot data as points

plt.xlabel('x')
plt.ylabel('y')

plt.legend()
```

When I ran this code I got.

```
Fitting model y=Ax^b
A=3.69+/-2.09
b=2.41+/-0.03
```

Nicely this is consistent with my input model.

4.2 Non-linear fitting

Sometimes you want to fit a non-linear model to your data. Here there is no analytically solution to getting a fit. That said there are many numerical methods that can get these fits through iteration (think loops).

```python
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as rnd

def sinfunc(x, p0, p1, p2, p3):
    return p0 + p1*np.sin(p2*(x-p3))

# Generate some fake data
Datax=rnd.random(50)*2*np.pi # start with 50 uniform points

# Set the 'true' solution to y=2*sin(x-0.1)
Datay= 2*np.sin(Datax-0.1)



# add noise with a standard deviation of 0.25 to the data
Datay+=0.25*rnd.randn(50)

from scipy.optimize import curve_fit # load non-linear fitting function
p, pcov = curve_fit(sinfunc, Datax, Datay) # perform fit

# p contains the fit
# pcov contains the covariance matrix

# print result
print('Fitting model y=p0 + p1*sin(p2*(x-p3))')
print('p0={:0.2f}+/-{:0.2f}'.format(p[0],2*pcov[0,0]))
print('p1={:0.2f}+/-{:0.2f}'.format(p[1],2*pcov[1,1]))
print('p2={:0.2f}+/-{:0.2f}'.format(p[2],2*pcov[2,2]))
print('p3={:0.2f}+/-{:0.2f}'.format(p[3],2*pcov[3,3]))

Plotx=np.linspace(0,2*np.pi) # create an x array for plotting the fit
Ploty=sinfunc(Plotx,p[0],p[1],p[2],p[3])

# plot Solution
plt.figure()
plt.plot(Datax,Datay,'b.',label='Data') #plot data as points
plt.plot(Plotx,Ploty,'r-', label='Fit') #plot data as points

plt.xlabel('x')
plt.ylabel('y')

plt.legend()
```

When I ran this code I got.

```
Fitting model y=p0 + p1*sin(p2*(x-p3))
p0=0.02+/-0.00
p1=2.05+/-0.01
p2=1.00+/-0.00
p3=0.05+/-0.01
```

The fit values are fairly good but the error estimates are clearly too low. This is not uncommon for fitting routines so be careful how you interpret your results (and other peoples).

4.3 Spline fitting and interpolation

Sometimes you are less interested in fitting a physical model and just need to interpolate between data points. For this spline curves are often a good solution.

In the example below I sparsely sample a sin curve, then compare a linear and spline interpolation.

```python
import matplotlib.pyplot as plt
import numpy as np


# Sample a sin curve
Datax=np.linspace(0,2*np.pi,4) # start with 50 uniform points

# Create a complex y data set
Datay= 2*np.sin(Datax-0.1)+2*Datax


import scipy.interpolate as interp

Plotx=np.linspace(0,2*np.pi,100) # create an x array for plotting the fit

Lineary=np.interp(Plotx,Datax,Datay)
Spliney=interp.spline(Datax,Datay,Plotx)
Truey= 2*np.sin(Plotx-0.1)+2*Plotx


# plot Solution
plt.figure()

plt.plot(Plotx,Truey,'b-', label='True Fit')
plt.plot(Plotx,Lineary,'r--', label='Linear Fit')
plt.plot(Plotx,Spliney,'g--', label='Spline Fit')

plt.plot(Datax,Datay,'b.',label='Data') #plot data as points


plt.xlabel('x')
plt.ylabel('y')

plt.legend()
```

Note that spline fits tend to assume things are very smooth. If your data is not smooth this is not going to give you a good interpolation!