Modeling Conditional Distributions with Neural Networks

October 27, 2016

```
In [2]: %matplotlib inline
        import matplotlib
        import seaborn as sns
        matplotlib.rcParams['savefig.dpi'] = 2 * matplotlib.rcParams['savefig.dpi']
```

1 Modeling Conditional Distributions with Neural Networks

1.0.1 Brian Goldman

1.1 Introduction

The goal of most supervised machine learning algorithms is to approximate a conditional expectation function– that is, to find the average value of a target variable given the values of several input variables. Just as the mean $\bar{X}$ of a set of values X minimizes the sum of squared error $\sum_{i=1}^{N}(x_i - c)^2$,

$$\frac{d}{dc}\sum_{i=1}^{N}(x_i - c)^2 = -2c\sum_{i=1}^{N}(x_i - c) = 0$$

implies

$$\sum_{i=1}^{N} x_i = Nc$$

and so

$$c = \bar{X},$$

we can observe that the conditional expectation function minimizes the sum of squared error for a regression problem: it will suffice to repeat the above calculation for each possible selection of the input variables. For a classification problem, the conditional expecation of a binary target variable X is equal to the probability $X = 1$.

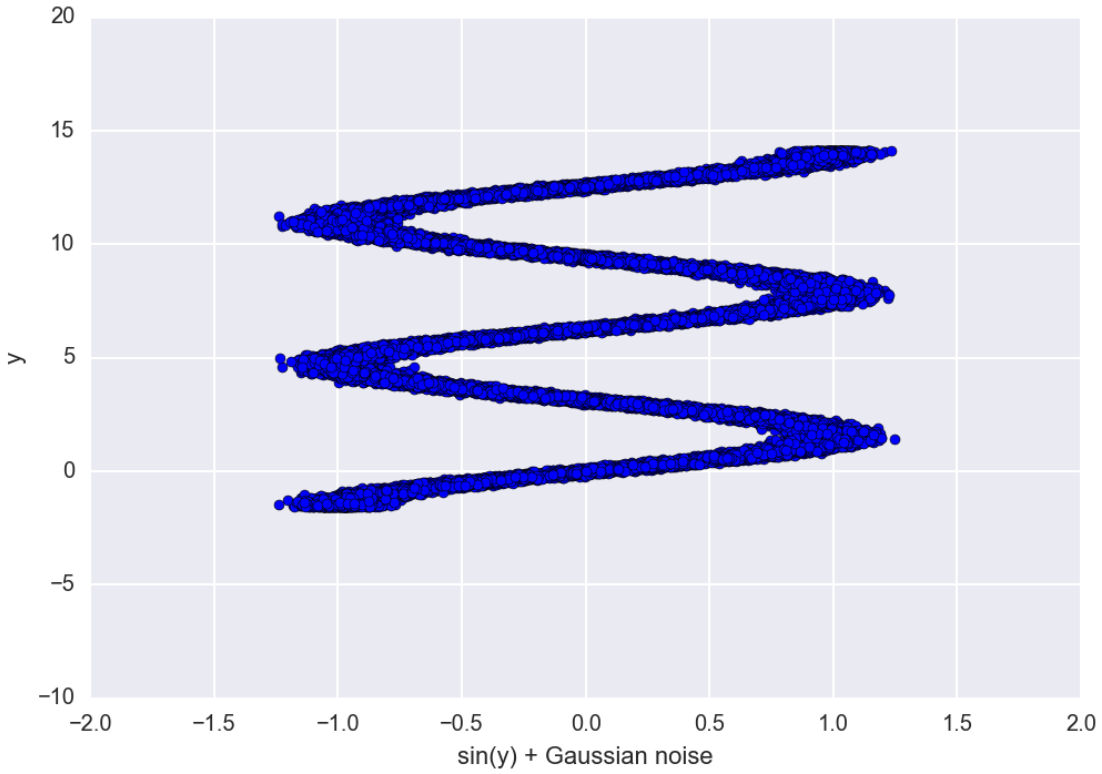
Modeling the conditional expectation function is hard (in particular, it requires a lot of data), and many algorithms make simplying assumptions at the cost of reduced accuracy. Ordinary linear regression, for example, can only find the best linear approximation of the conditional expectation function.

Still, in some cases, we may not be satisfied with the conditional expectation function, but we may prefer to know the conditional density function of our target variable given the values of our input variables. This may be particularly valuable if the conditional distribution is multimodal. For example, given an anonymous customer's online purchase history, we might predict several possible income ranges corresponding to different demographic profiles. In many such cases, a conditional average simply won't do. We'd much rather know how a customer splits their time between New York City and Miami than observe that, on average, they live in Charleston, SC.

1.2 An Example Problem

Let's generate a sample problem via simulation. 50000 points y_i are randomly drawn from the range $[-\pi/2, 9\pi/2]$. For each point, we will set $x_i = sin(y_i) + N_i$, where N_i is some small normally distributed error. Our goal is to model the distribution of the y values given an x value.

In [3]:
```
import numpy as np
import scipy
import sklearn
import matplotlib.pyplot as plt

y=np.pi*(5*np.random.rand(50000,1)-.5)
x=np.sin(y)+np.random.normal(0,.07,(50000,1))
plt.xlabel('sin(y) + Gaussian noise')
plt.ylabel('y')
plt.scatter(x, y)
axes = plt.gca()
axes.set_xlim([-2,2])
axes.set_ylim([-10,20])
plt.show()
```

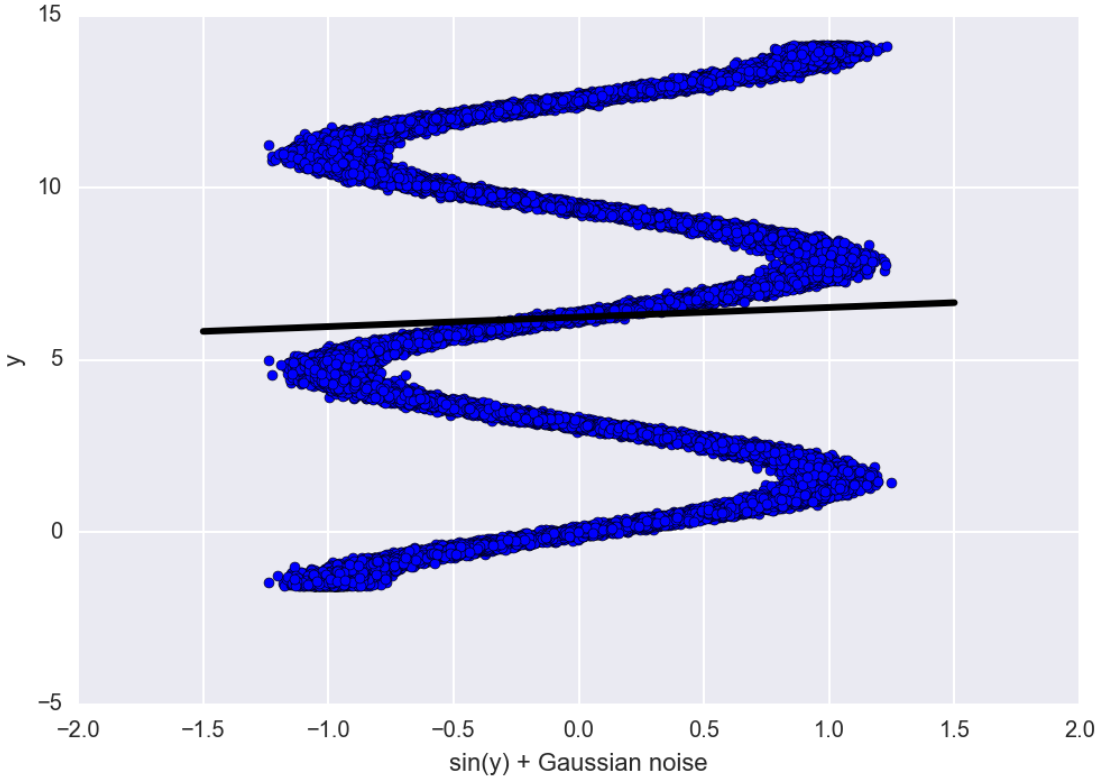


1.2.1 The Conditional Expectation Function

As a first check, we might try modeling the conditional expectation function using simple machine learning techniques, such as a linear regression (line of best fit shown in black):

In [7]:
```
from sklearn.linear_model import LinearRegression

ourLR=LinearRegression()
ourLR.fit(x,y)
plt.xlabel('sin(y) + Gaussian noise')
plt.ylabel('y')
```

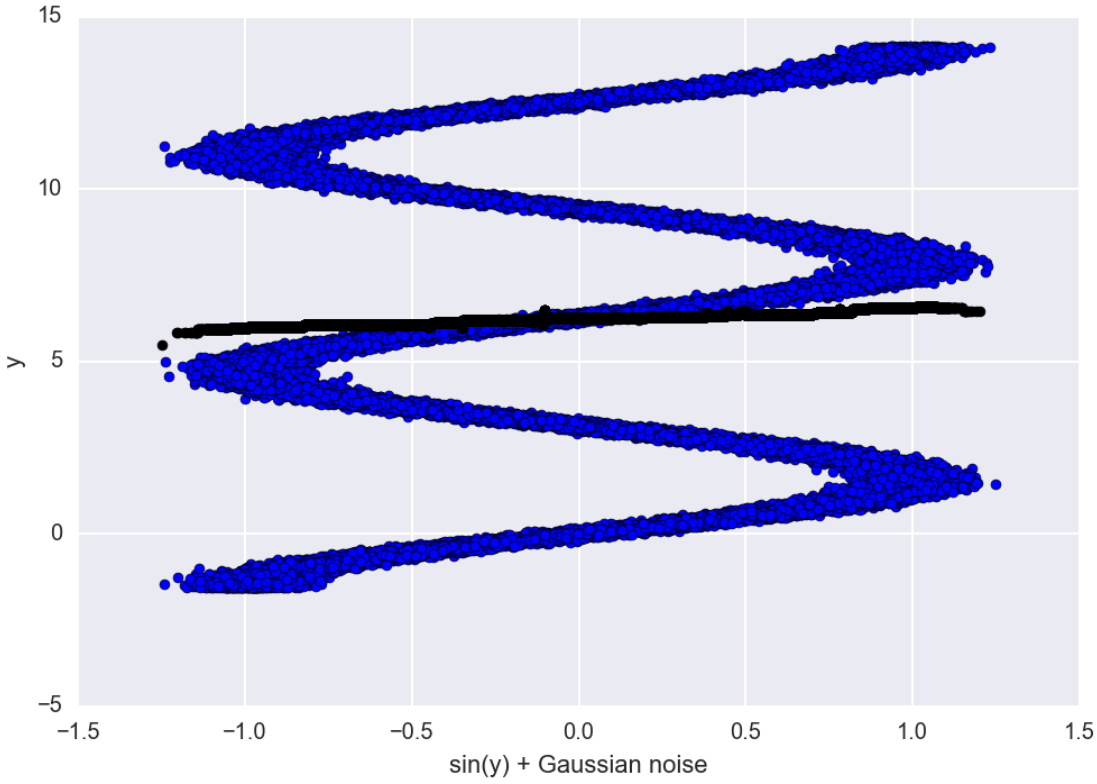
```
plt.scatter(x, y)
plt.plot([-1.5, 1.5], [ourLR.intercept_-1.5*ourLR.coef_[0], ourLR.intercept_+1.5*ourLR.coef_[0]]
axes.set_xlim([-2,2])
axes.set_ylim([-10,20])
plt.show()
```



Or a cross-validated random forest (predicted y values shown in black):

```
In [15]: from sklearn.ensemble import RandomForestRegressor
         from sklearn.grid_search import GridSearchCV
         import warnings

         warnings.filterwarnings("ignore")
         ourRF=GridSearchCV(RandomForestRegressor(),{'n_estimators':[10,20,40],'max_depth':[2,3,5,7]})
         ourRF.fit(x,np.ravel(y))
         plt.xlabel('sin(y) + Gaussian noise')
         plt.ylabel('y')
         plt.scatter(x, y)

         y_test=np.pi*(5*np.random.rand(10000,1)-.5)
         x_test=np.sin(y_test)+np.random.normal(0,.07,(10000,1))
         y_predicted=ourRF.predict(x_test)

         plt.scatter(x_test, y_predicted, color='black')

         axes.set_xlim([-2,2])
```

```
axes.set_ylim([-10,20])
plt.show()
```



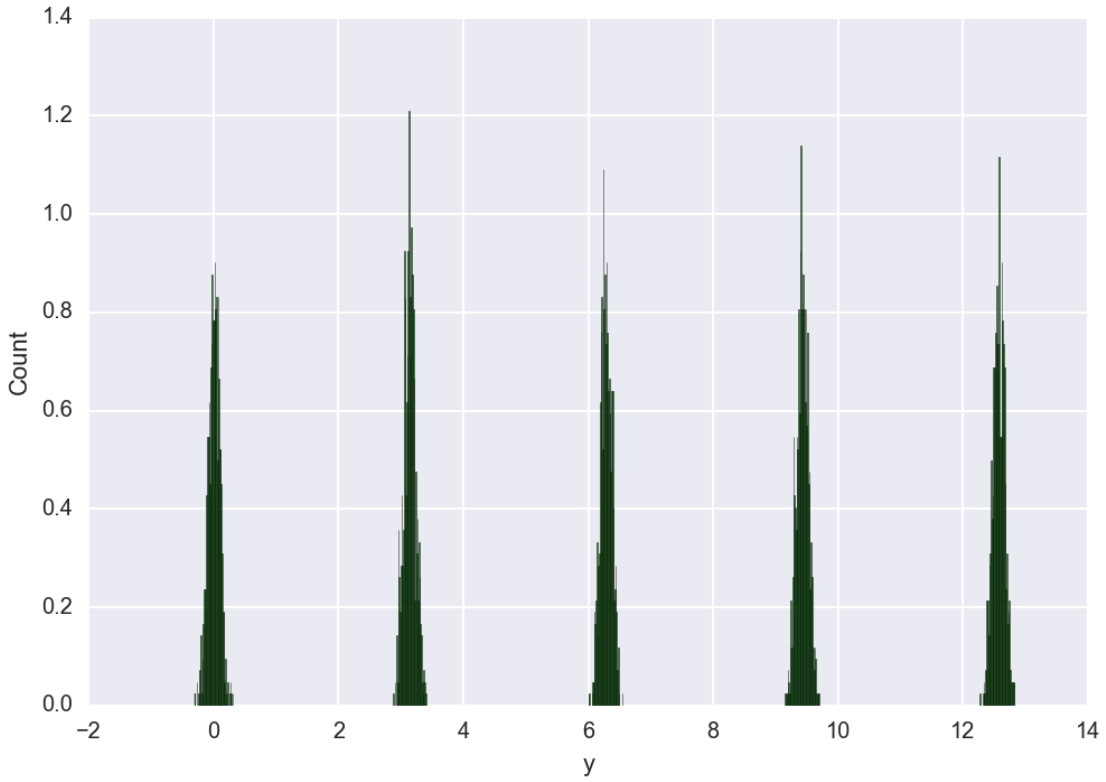
It's not hard to see why these techniques don't work; each can only return a single y value for a given x value, even though there are several plausible values of y for each x. We would do better to model the conditional density function of y given x– that is, the function that tells the probability of attaining a particular value y given a particular value of x.

1.2.2 Modeling Conditional Distributions

Let's look at the y values for a slice of our data, say, when x (approximately) equals zero:

```
In [9]: indices=[i for i in range(len(x)) if np.abs(x[i])<=.1]
        y_slice=y[indices]
        n, bins, patches=plt.hist(y_slice, 1000, normed=1, facecolor='green', alpha=0.75)
        plt.xlabel('y')
        plt.ylabel('Count')
```

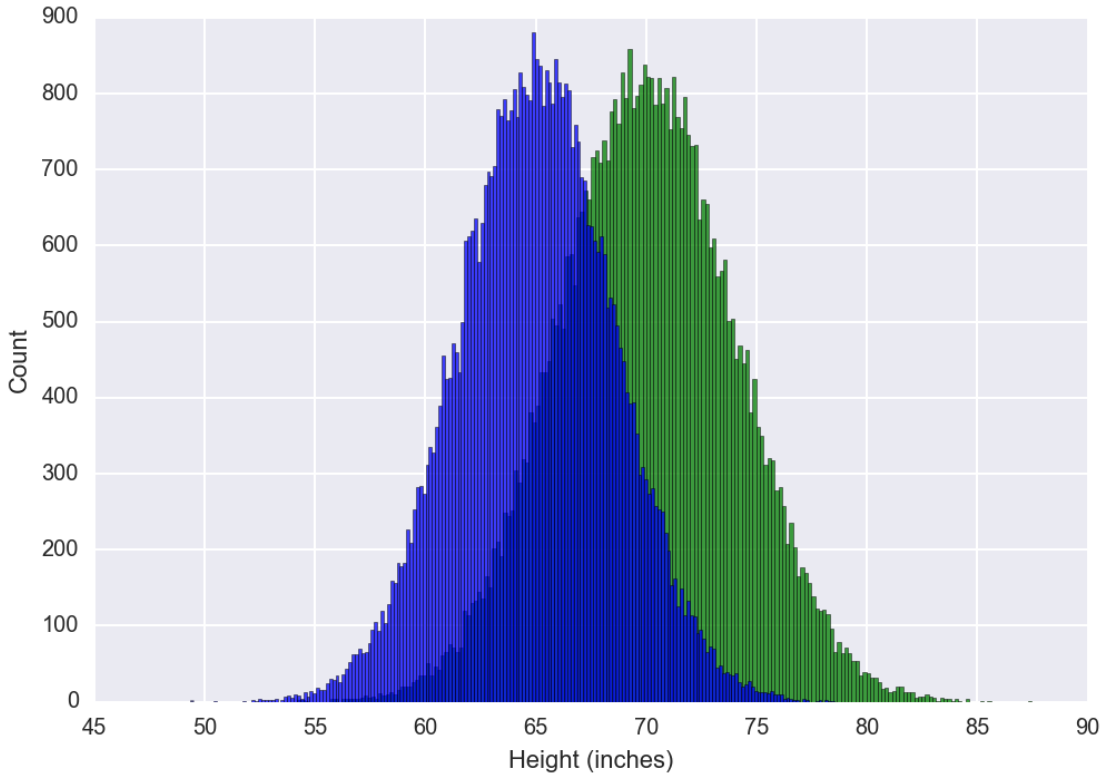
```
Out[9]: <matplotlib.text.Text at 0x1191c7350>
```



We can see that the data lies in 5 components, each approximately normally distributed around $n\pi$ for $n = 0 \ldots 4$. We will model the distribution of y given x as a Gaussian mixture model. A Gaussian mixture model is comprised of multiple Gaussian (or normal) variables, each with a sampling probability. An excellent example is human height. The average American man is 70 inches tall (standard deviation: 4 inches); the average American woman is 65 inches tall (standard deviation: 3.5 inches)

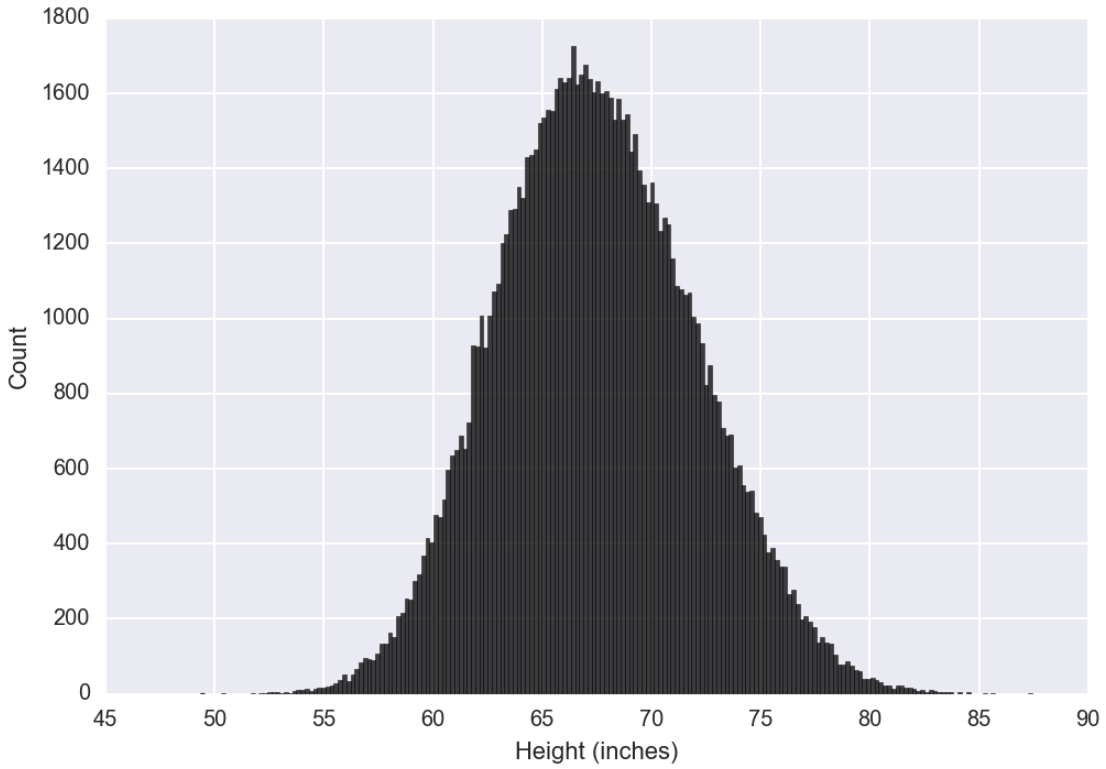
```
In [10]: number_men=np.random.binomial(100000, .5)
         men_heights=np.random.normal(70,4, number_men)
         women_heights=np.random.normal(65,3.5, 100000-number_men)
         n, bins, patches=plt.hist(men_heights, 200, normed=0, facecolor='green', alpha=0.75)
         n, bins, patches=plt.hist(women_heights, 200, normed=0, facecolor='blue', alpha=0.75)
         plt.xlabel('Height (inches)')
         plt.ylabel('Count')
```

```
Out[10]: <matplotlib.text.Text at 0x118408a50>
```



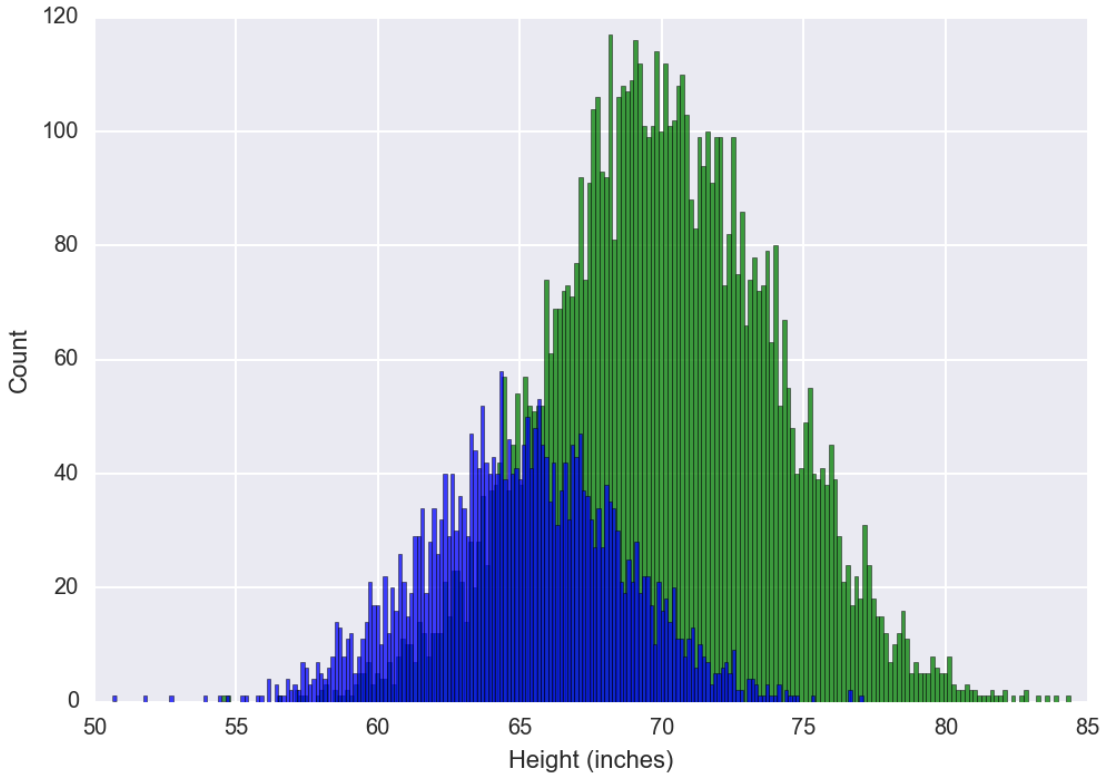
Combining men and women into one category, we plot Americans' heights:

```
In [11]: heights=np.concatenate((men_heights, women_heights))
         n, bins, patches=plt.hist(heights, 200, normed=0, facecolor='black', alpha=0.75)
         plt.xlabel('Height (inches)')
         plt.ylabel('Count')

Out[11]: <matplotlib.text.Text at 0x11820f210>
```



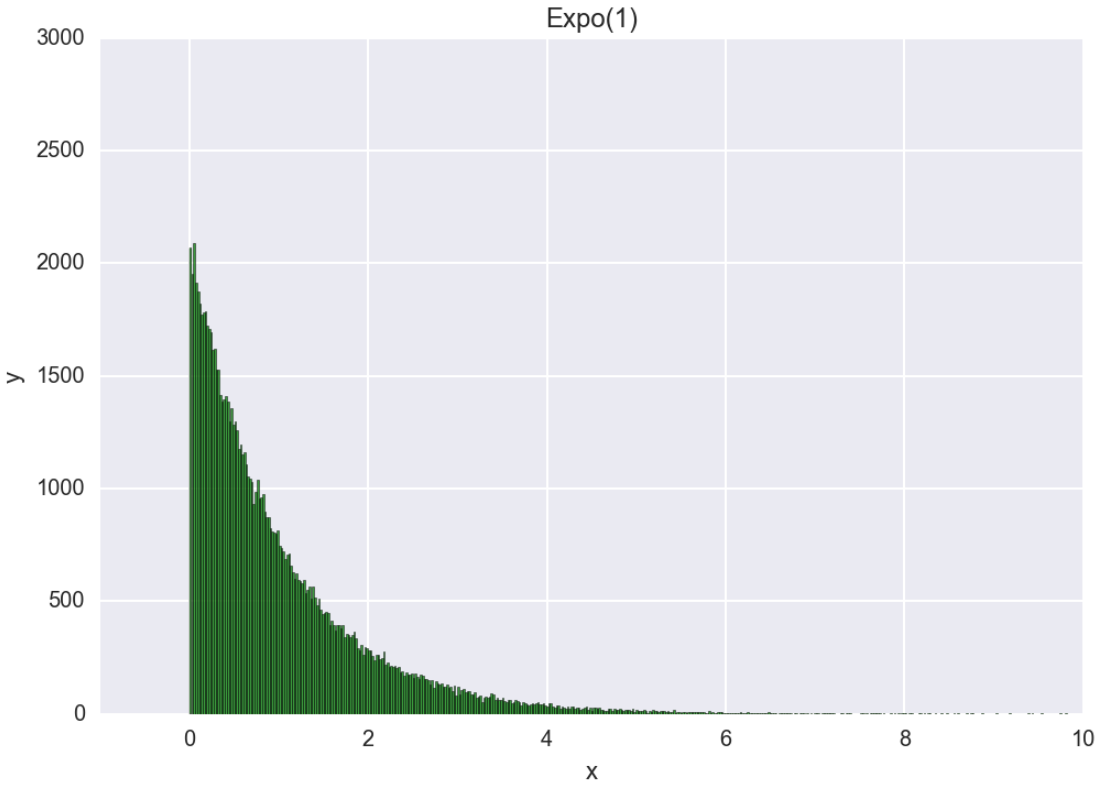
This is a Gaussian mixture model where one component has mean 70, standard deviation 4, and sampling probability 0.5; and the other component has mean 65, standard deviation 3.5, and sampling probability 0.5. That it appears roughly normal is coincidence; if the means were further apart, we would observe clear bimodality. There's no need for the sampling probabilities to be equal, though. Let's try oversampling men, so that the male sampling probability is 0.7 and the female sampling probability is 0.3:

```
In [12]: number_men=np.random.binomial(10000, .7)
         men_heights=np.random.normal(70,4, number_men)
         women_heights=np.random.normal(65,3.5, 10000-number_men)
         n, bins, patches=plt.hist(men_heights, 200, normed=0, facecolor='green', alpha=0.75)
         n, bins, patches=plt.hist(women_heights, 200, normed=0, facecolor='blue', alpha=0.75)
         plt.xlabel('Height (inches)')
         plt.ylabel('Count')

Out[12]: <matplotlib.text.Text at 0x11ef88650>
```



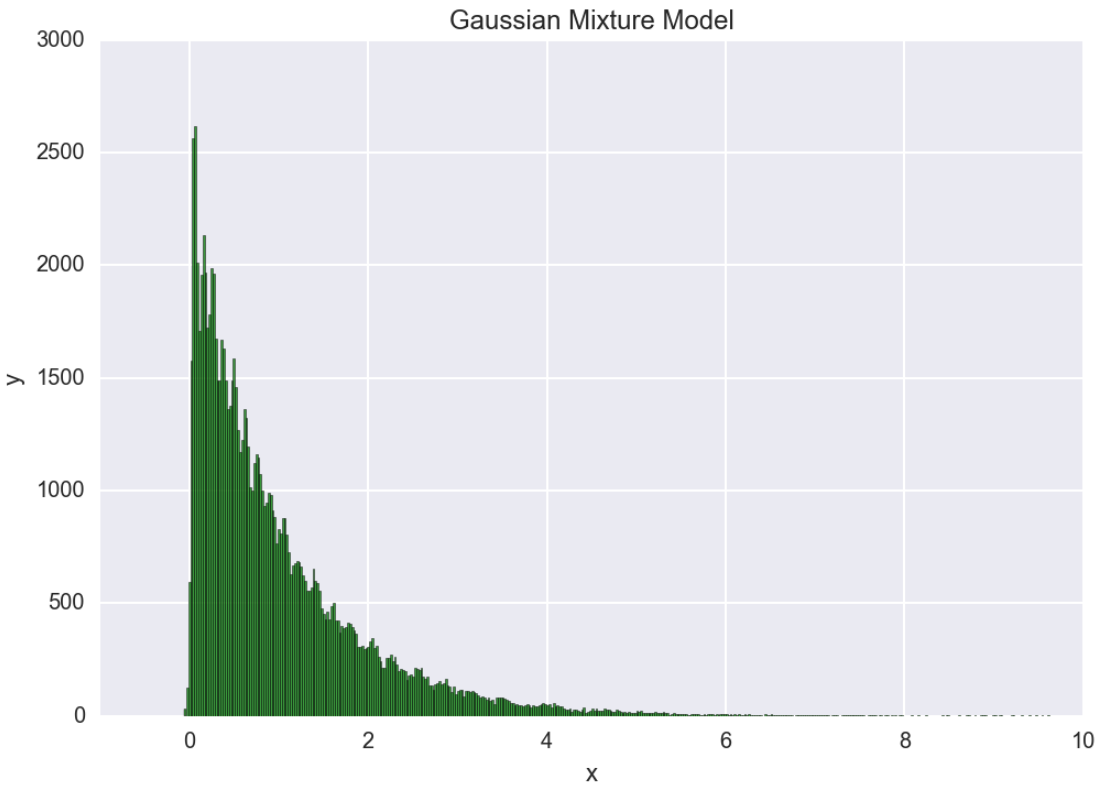
Although a Gaussian mixture model always is comprised of normal components, granted enough components, a Gaussian mixture model can replicate any probability distribution. We observe that a Gaussian mixture model can replicate any discrete distribution (allow the number of components to equal the number of possible outcomes in the distribution; set the means of the components equal to those outcomes; and let the standard deviations approach zero). And, of course, any continuous distribution can be approximated using discrete distributions. Below, we plot a histogram of data sampled from an exponential distribution with mean 1:

```
In [43]: exp_variates=np.random.exponential(1, (100000,1))
         n, bins, patches=plt.hist(exp_variates, 500, normed=0, facecolor='green', alpha=0.75)
         plt.xlabel('x')
         plt.ylabel('y')
         plt.title('Expo(1)')
         axes = plt.gca()
         axes.set_xlim([-1,10])
         axes.set_ylim([0,3000])

Out[43]: (0, 3000)
```



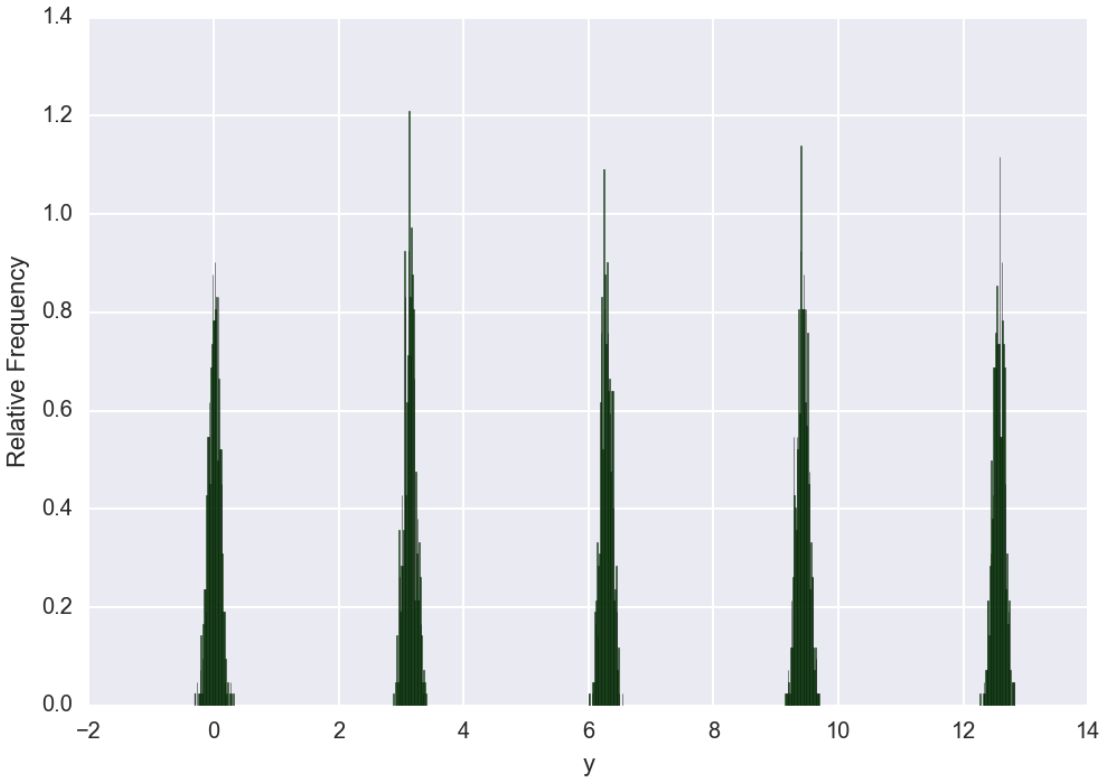
And a Gaussian mixture model with 25 components fitted to the same data:

```
In [44]: from sklearn.mixture import GaussianMixture

         plt.xlabel('x')
         plt.ylabel('y')
         plt.title('Gaussian Mixture Model')
         expGMM=GaussianMixture(n_components=25)
         expGMM.fit(exp_variates)
         expGMM_output=expGMM.sample(100000)
         n, bins, patches=plt.hist(expGMM_output[0], 500, normed=0, facecolor='green', alpha=0.75)
         axes = plt.gca()
         axes.set_xlim([-1,10])
         axes.set_ylim([0,3000])

Out[44]: (0, 3000)
```



Now, let's try fitting a Gaussian mixture model to the conditional distribution of y given $x \approx 0$ from our example problem:

```
In [13]: n, bins, patches=plt.hist(y_slice, 1000, normed=1, facecolor='green', alpha=0.75)
         plt.xlabel('y')
         plt.ylabel('Relative Frequency')

Out[13]: <matplotlib.text.Text at 0x11ff461d0>
```



The scikit-learn Python package implents Gaussian mixture model fitting for us, but we'll be on our own soon, so it will be useful to understand how it works. Let's start be choosing a number of mixture components for our model; here, let's agnostically pick 20. Given our sample data Y, we'll define a *likelihood* function $L(\theta)$:

$$L(\theta) = p(Y|\theta) = \prod_{i=1}^{N} \sum_{j=1}^{20} \frac{\alpha_j}{\sigma_j \sqrt{2\pi}} \; e^{-\frac{(y_i - \mu_j)^2}{2\sigma_j^2}}$$

Here, $\alpha_j, \mu_j, \sigma_j$ are the sampling probability, mean, and standard deviation, respectively, of component j. $L(\theta) = p(Y|\theta)$ is the probability of observing the data that we have, given a particular setting of these model parameters. Applying Bayes' Law, we see that the probability $p(\theta|Y)$ of a particular set of model parameters given our data increases proportionally to $L(\theta)p(\theta)$, where $p(\theta)$ is the prior probability of that set of model parameters. If we assume a uniform prior, maximizing $p(\theta|Y)$ is equivalent to maximizing $L(\theta)$.

In fact, we will find the parameters that minimize

$$-\log L(\theta) = -\sum_{i=1}^{N} \log \Big(\sum_{j=1}^{20} \frac{\alpha_j}{\sigma_j \sqrt{2\pi}} \; e^{-\frac{(y_i - \mu_j)^2}{2\sigma_j^2}} \Big)$$

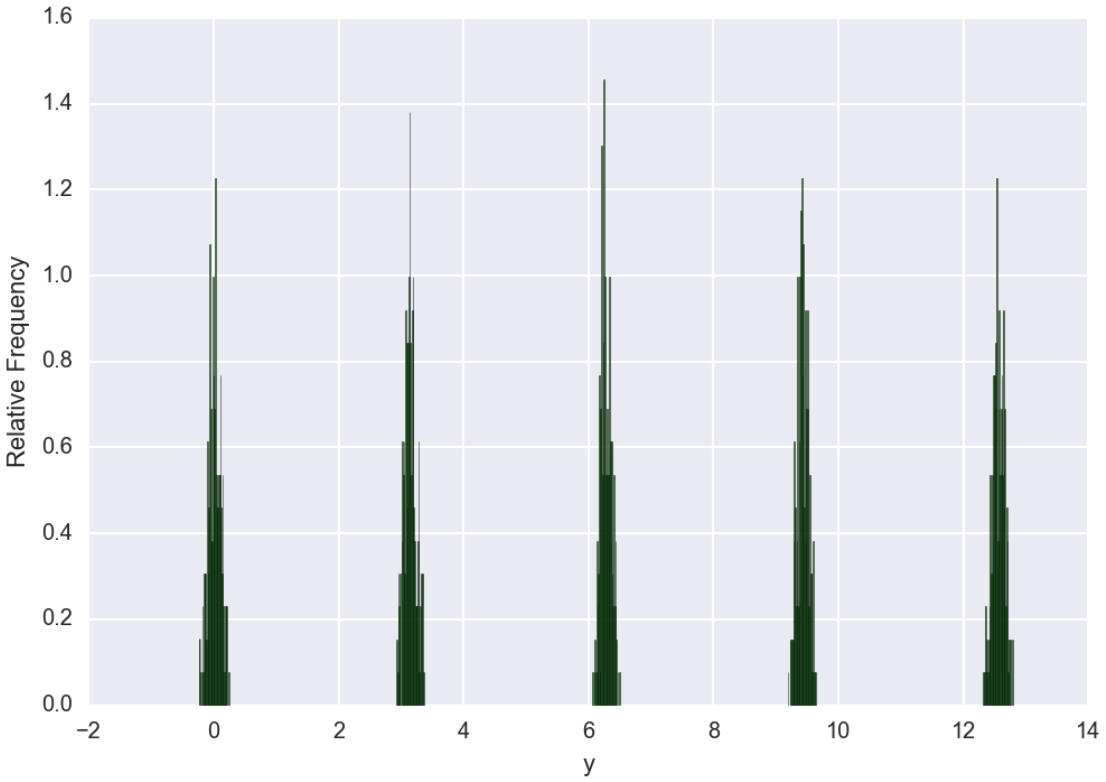
.

Since $\log(x)$ is a monotonic function, this will yield the same parameters, though this form is generally more tractable to analytic methods.

$-\log L(\theta)$ can be minimized using a variety of non-linear optimization methods. The implementation in scikit-learn is via expectation maximization– an algorithm related to k-means clustering, though, when we're building our own model, we'll use gradient methods.

Below, we fit a Gaussian mixture model to our data, then draw new data from the fitted model to see if it replicates our original distribution. Note that we chose n_components=20 for concreteness; we may also choose to cross-validate on the number of components.

```
In [14]: from sklearn.mixture import GaussianMixture

         ourGMM=GaussianMixture(n_components=20)
         ourGMM.fit(y_slice)
         GMM_output=ourGMM.sample(1000)
         n, bins, patches=plt.hist(GMM_output[0], 1000, normed=1, facecolor='green', alpha=0.75)
         plt.xlabel('y')
         plt.ylabel('Relative Frequency')

Out[14]: <matplotlib.text.Text at 0x1214e2fd0>
```



Unfortunately, we need a conditional density function for every possible setting of the input variables (in this case, every possible x value). Our key idea will be to fit a (generally non-linear) function from the input variables to the Gaussian mixture model parameters. Once the function is fit, we can immediately compute the mixture model parameters for given inputs and determine the corresponding density function.

We can fit this function as a neural network. For our example problem, the network will have three output nodes for each Gaussian mixture component– one each representing the component mean, standard deviation, and sampling probability. For component j, allow output node $z_{j\mu}$ to correspond to the mean, $z_{j\sigma}$ to the standard deviation, and $z_{j\alpha}$ the sampling probability. Since $\sigma_j > 0$, we will set $\sigma_j = e^{z_{j\sigma}}$. Likewise, since $\sum_{j=1}^{C} \alpha_j = 1$ and $\alpha_j > 0$, we will set

$$\alpha_j = \frac{e^{z_{j\alpha}}}{\sum_{j=1}^{C} e^{z_{j\alpha}}},$$

where C is the number of mixture components. This latter transformation is known as the softmax, and it appears frequently in the theory of linear classifiers. Our objective function remains much as it did before, except now the Gaussian mixture parameters are functions of the inputs:

$$-\log L(\theta) = -\sum_{i=1}^{N} \log\Big(\sum_{j=1}^{C} \frac{\alpha_j(x)}{\sigma_j(x)\sqrt{2\pi}} \; e^{-\frac{(y_i-\mu_j(x))^2}{2\sigma_j(x)^2}}\Big)$$

We wish find the functional mapping (parameterized by the weights of our neural network) from the input variables to the outputs $z_{j\mu}$, $z_{j\sigma}$, and $z_{j\alpha}$ which minimize this objective function. Below we define a class which searches for this functional mapping. Given a current setting of the neural network weights, we find the gradient of the objective function in weight space using standard backpropagation techniques, adjusted both for our particular objective funtion and for the transformations we applied to the model outputs. A detailed discussion of this procedure can be found in Christopher Bishop's Neural Networks for Pattern Recognition (https://www.amazon.com/Networks-Pattern-Recognition-Advanced-Econometrics/dp/0198538642); see especially sections 4.8 and 6.4 (mostly accessible in the book preview if you search for the corresponding headings). We then seek a minimum of our objective function using stochastic gradient descent and line search.

In [5]:
```
import copy
import time



class Neural_Network_Multimodal:

    def __init__(self, layers, hidden, inpt, outpt, kernels):
        #@layers is the number of hidden layers; @hidden the number of nodes per hidden layer;
        #input features; @outpt the number of output features (1 in sample problem); @kernels t
        self.layers=layers
        self.hidden=hidden
        self.inpt=inpt
        self.outpt=outpt
        self.kernels=kernels
        self.node_count=inpt+layers*hidden+(outpt+2)*kernels
        nodes=range(self.node_count)
        self.nodes=[nodes[0:inpt]]+[nodes[x:x+hidden] for x in xrange(inpt, inpt+hidden*layers,
        self.weights=[(np.random.rand(len(self.nodes[i])+1,len(self.nodes[i+1]))-.5) for i in ra

    def find_Gradient(self, sample_in, sample_out):
        #This method finds the gradient of the error function for a single observation
        #The full gradient can be found by calling this function on all observations and adding
        current=sample_in
        deltas=[]
        activations=[]
        #Foreward-propagate input data to find activations of hidden-layer nodes and output nod
        #Hidden-layer nodes transformed using tanh function
        for k in range(self.layers+1):
            activations.append(current)
            current=np.append(current,[1])
            current=np.dot(current,self.weights[k])
            if k!=self.layers:
                current=np.tanh(current)
        activations.append(current)
        #@alphas and sigmas as documented above
        alphas=[]
        sigmas=[]
        pis=[]
```

```
        residual_comps=[]
        sum_sq=[]
        delta_out=np.zeros((self.outpt+2)*self.kernels)

        for i in range(self.kernels):
            #@current gives z_j\mu, z_j\sigma, and z_j\alpha for the current mixture component
            current=activations[self.layers+1][(self.outpt+2)*i:(self.outpt+2)*(i+1)]
            #@residual_comps gives the difference from the component mean to the sample output
            residual_comps.append(current[0:self.outpt]-np.array(sample_out))
            #@sum_sq gives the squared distance from the component mean to the sample output
            sum_sq.append(np.linalg.norm(residual_comps[i])**2)
            sigmas.append(10*np.exp(current[self.outpt]))
            alphas.append(np.exp(current[self.outpt+1]))
        sum_exp=sum(alphas)
        for i in range(self.kernels):
            alphas[i]=alphas[i]/sum_exp
            pis.append(alphas[i]*np.exp(-sum_sq[i]/(2*(sigmas[i]**2)))/(((2*np.pi)**(self.outpt/
        sum_pialphs=sum(pis)
        for i in range(self.kernels):
            #@pis give the probability of the sample observation having originated from each ke
            pis[i]=pis[i]/sum_pialphs
        for i in range(self.kernels):
            #@d_z_* gives the derivative of the error function with respect to the values of th
            d_z_alpha=alphas[i]-pis[i]
            d_z_sigma=-pis[i]*((sum_sq[i]/(sigmas[i]**2))-self.outpt)
            d_z_mu=(pis[i]/(sigmas[i]**2))*residual_comps[i]
            #@delta_out is a vector of these same derivatives
            delta_out[(self.outpt+2)*i:((self.outpt+2)*(i+1))-2]=d_z_mu
            delta_out[((self.outpt+2)*(i+1))-2]=d_z_sigma
            delta_out[((self.outpt+2)*(i+1))-1]=d_z_alpha
        #@delta will store the derivatives of the error function w.r.t. the value of output and
        deltas.append(delta_out)
        for k in xrange(self.layers,0,-1):
            #derivatives w.r.t earlier-layer nodes are calculated using derivatives w.r.t. late
            w_d=np.dot(self.weights[k][0:-1],deltas[0])
            new_deltas=np.array([(1-activations[k][i]**2)*w_d[i] for i in range(len(w_d))])
            deltas.insert(0,new_deltas)
        gradient=[]
        for k in range(self.layers+1):
            #derivatives w.r.t. weights calculated using derivatives w.r.t. node values
            gradient.append(np.outer(np.insert(activations[k],len(activations[k]),1),deltas[k]))
        return gradient


    def gradient_Fit(self, samples_in, samples_out, t=10, eta=10, eta_increment=.1):
        #This method performs stochastic gradient descent for t seconds.  The whole data set sh
        #entered as samples_in, samples_out.
        #@eta determines the size of the step in the direction of the negative gradient.
        start = time.clock()
        while time.clock()-start<t:
            order=np.random.permutation(len(samples_in))
            for j in order:
                g=self.find_Gradient(samples_in[j],samples_out[j])
                eta+=eta_increment
```

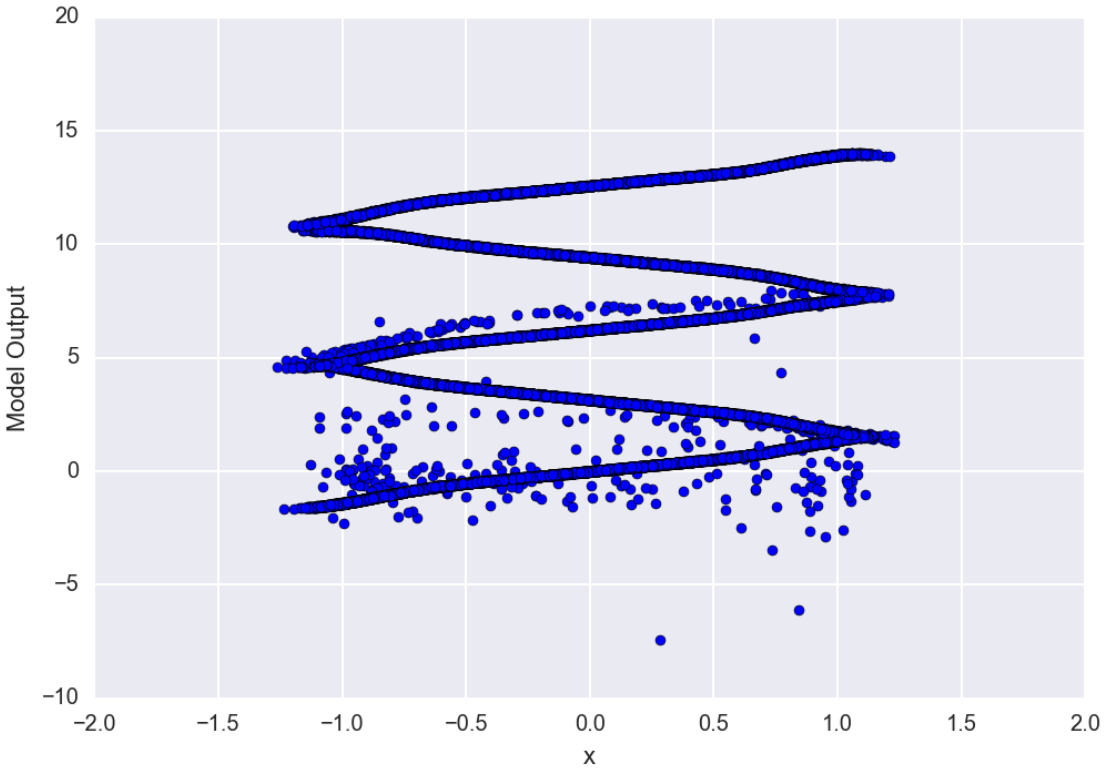
```
            for k in range(len(g)):
                self.weights[k]=self.weights[k]-(1.0/eta)*g[k]


    def evaluate_Error(self, sample_in, sample_out):
        #This method evaluates the objective function for a single observation and the current
        #To find the total objective function, apply to all observations and add the results
        #Variables as described in @find_gradient
        current=sample_in
        deltas=[]
        activations=[]
        for k in range(self.layers+1):
            activations.append(current)
            current=np.append(current,[1])
            current=np.dot(current,self.weights[k])
            if k!=self.layers:
                current=np.tanh(current)
        activations.append(current)
        alphas=[]
        sigmas=[]
        residual_comps=[]
        sum_sq=[]
        pis=[]
        for i in range(self.kernels):
            current=activations[self.layers+1][(self.outpt+2)*i:(self.outpt+2)*(i+1)]
            residual_comps.append(current[0:self.outpt]-np.array(sample_out))
            sum_sq.append(np.linalg.norm(residual_comps[i])**2)
            sigmas.append(np.exp(current[self.outpt]))
            alphas.append(np.exp(current[self.outpt+1]))
        sum_exp=sum(alphas)
        for i in range(self.kernels):
            alphas[i]=alphas[i]/sum_exp
            pis.append(alphas[i]*np.exp(-sum_sq[i]/(2*(sigmas[i]**2)))/(((2*np.pi)**(self.outpt/
        sum_pialphs=sum(pis)
        return -np.log(sum_pialphs)


    def gradient_LS_Fit(self, samples_in, samples_out, score_count=500, t=10, initial_coef=.000
        #This methods determines the gradient of the objective function defined by a random bat
        #Of size @score-count, searches for a minimum of that objective function along the dire
        #Then selects a new batch, for t seconds.  Set score_count=0 to use the whole data set
        count=0
        if score_count==0:
            score=float('inf')
        population=score_count
        start = time.clock()
        while time.clock()-start<t:
            #initialize gradient
            g=[]
            for k in self.weights:
                g.append(np.zeros(k.shape))
            order=np.random.permutation(len(samples_in))
            for j in order[0:population]:
                h=self.find_Gradient(samples_in[j],samples_out[j])
```

```
                for k in range(len(self.weights)):
                    g[k]=g[k]+h[k]

            improve=True
            count+=1
            coef=initial_coef/(count+1)

            #reset objective function if we have selected a new batch of observations
            if score_count!=0:
                score=float('inf')

            #if the current step-size in the negative gradient direction is an improvement; dou
            #otherwise, return to the previous step-size
            while improve==True:
                for k in range(len(self.weights)):
                    self.weights[k]=self.weights[k]-coef*g[k]
                new_score=0
                if score_count!=0:
                    order=order[0:score_count]
                for j in order:
                    new_score+=self.evaluate_Error(samples_in[j],samples_out[j])
                if new_score<score:
                    score=new_score
                    coef*=2
                else:
                    improve=False
                    for k in range(len(self.weights)):
                        self.weights[k]=self.weights[k]+coef*g[k]




    def rand_predict(self, sample_in):
        #For a given setting of the inputs, this method computes the Gaussian mixture model par
        #Then draws a random sample from the corresponding distribution.
        #Variables as described in @find_gradient
        current=sample_in
        for k in range(self.layers+1):
            current=np.append(current,[1])
            current=np.dot(current,self.weights[k])
            if k!=self.layers:
                current=np.tanh(current)

        alphas=[]
        sigmas=[]
        means=[]
        sum_sq=[]
        old=current
        for i in range(self.kernels):
            current=old[(self.outpt+2)*i:(self.outpt+2)*(i+1)]
            means.append(current[0:self.outpt])
            sum_sq.append(np.linalg.norm(means[i])**2)
            sigmas.append(np.exp(current[self.outpt]))
```

```
            alphas.append(np.exp(current[self.outpt+1]))
        sum_exp=sum(alphas)
        for i in range(self.kernels):
            alphas[i]=alphas[i]/sum_exp
        class_prob=np.random.rand()
        class_val=0
        for i in range(self.kernels):
            class_prob-=alphas[i]
            if class_prob<=0.0:
                class_val=i
                break
        out_vec=np.zeros(self.outpt)
        for j in range(self.outpt):
            out_vec[j]=np.random.normal(means[class_val][j],sigmas[class_val])
        return out_vec
```

In [6]:
```
network=Neural_Network_Multimodal(layers=6,hidden=30,inpt=1,outpt=1,kernels=30)

#This will train for 2 minutes 40 seconds
#See html for sample output

network.gradient_LS_Fit(x, y, t=80)
network.gradient_Fit(x, y, t=80)


y_source=np.pi*(5*np.random.rand(50000,1)-.5)
x_plot=np.sin(y_source)+np.random.normal(0,.07,(50000,1))
y_plot=np.array([network.rand_predict(x_val) for x_val in x_plot])

plt.scatter(x_plot, y_plot)
```

```
axes = plt.gca()
axes.set_xlim([-2,2])
axes.set_ylim([-10,20])
plt.xlabel('x')
plt.ylabel('Model Output')
plt.show()
```



1.3 Conclusion

How well does your plot match the original data? How does it compare to the outputs from our linear regression and random forest models? Your results may vary in quality; non-linear optimization is inherently slow and imprecise. See the html document for a relatively good outcome.

There are many additions you may conceive of to our model. You might cross-validate over the parameters (e.g. the structure of the neural network, the training time, the batch size in the line search method). Since training is slow, we provide you with a set of parameters we've found successful. You could apply a penalty function to limit the number of active Gaussian mixture components at any given setting of the paremeters, or to limit the magnitude of the weights in the neural network. And you may try implementing other non-linear optimization methods, such as conjugate gradient or BFGS.

We've written the class Neural_Network_Multimodal to generalize to multi-dimensional outputs. In this case, we've constrained the covariance matrices of the mixture components to be scalar multiples of the identity matrix. This poses a trade-off: it reduces the generality of each component, but allows us to train more components at a given time. Theoretically, such a Gaussian mixture model with sufficiently many components can still replicate any multivariate probability density function– for reasons much the same as we described for the univariate case.