CS109A Introduction to Data Science

Homework 6: Multilayer Feedforward Network - Dealing with Missing Data

Harvard University
Fall 2018
Instructors: Pavlos Protopapas, Kevin Rader

In [2460]:
```
#RUN THIS CELL
import requests
from IPython.core.display import HTML
styles = requests.get("https://raw.githubusercontent.com/Harvard-IACS/2018-CS109A/
HTML(styles)
```

Out[2460]:

INSTRUCTIONS

- To submit your assignment follow the instructions given in canvas (https://canvas.harvard.edu/courses/42693/pages/homework-policies-and-submission-instructions).
- This homework can be submitted in pairs.
- If you submit individually but you have worked with someone, please include the name of your **one** partner below.

Names of person you have worked with goes here:

In [2461]:
```
%matplotlib inline
import numpy as np
import numpy.random as nd
import pandas as pd
import math
import matplotlib.pyplot as plt
import matplotlib as mpl

import os
import seaborn as sns
sns.set(style="darkgrid")

from sklearn.linear_model import LogisticRegressionCV
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import Imputer
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from IPython.display import display
import statsmodels.api as sm
from statsmodels.regression.linear_model import OLS
```

In [2462]:
```
mpl.rcParams['axes.prop_cycle'] = mpl.cycler(color=['b', 'r', 'm','g'])
```

Overview

In this homework, you are free to explore different ways of solving the problems -within the restrictions of the questions. Your solutions should read like a report with figures to support your statements. Please include your code cells as usual but augment your solutions with written answers. We will also check for code readability and efficiency as we feel you have some experience now. In particular, for Q1, we expect you to write appropriate functions, such as your code can be generalized beyond the specified network architectures of his homework.

For this homework you may **not** use a machine learning library such as `keras` or `tensorflow` to build and fit the network. The objective is to build the network equations from scratch.

- Q1 explores approximating a function using a **Multilayer Feedforward Network** with one input layer, one hidden layer, and one output layer.
- Q2 deals with missing data in a medical dataset.

Question 1: Construct a feed forward neural network [50 pts]

In this part of the homework you are to construct three feed forward neural networks consisting of an input layer, one hidden layer with 1, 2 and 4 nodes respectively, and an output layer. The hidden layer uses the sigmoid as the activation function and use a linear output node. You should code the equations from scratch.

You are given three datasets containing (x, y) points where $y = f(x)$:

- In the first dataset, $f(x)$ is a **single step** function (data in `data/step_df.csv`),
- In the second dataset, $f(x)$ is a **one hump** function (data in `data/one_hump_df.csv`),
- In the third dataset, $f(x)$ is a **two equal humps** function (data in `data/two_hump_df.csv`).

1.1 Create a plot of each dataset and explore the structure of the data.

1.2 Give values to the weights **manually**, perform a forward pass using the data for the **single step** function and a hidden layer of **one** node, and plot the output from the network, in the same plot as the true y values. Adjust the weigths (again manualy) until the plots match as closely as possible.

1.3 Do the same for the **one hump** function data, this time using a hidden layer consisting of **two** nodes.

1.4 Do the same for the **two hump** function data but this time increase the number of hidden nodes to **four**.

1.5 Choose the appropriate loss function and calculate and report the loss from all three cases. Derive the gradient of the output layer's weights for all three cases (step, one hump and two humps). Use the weights for the hidden layers you found in the previous question and perform gradient descent on the weights of this layer (output layer). What is the optimised weight value and loss you obtained? How many steps did you take to reach this value? What is the threshold value you used to stop?

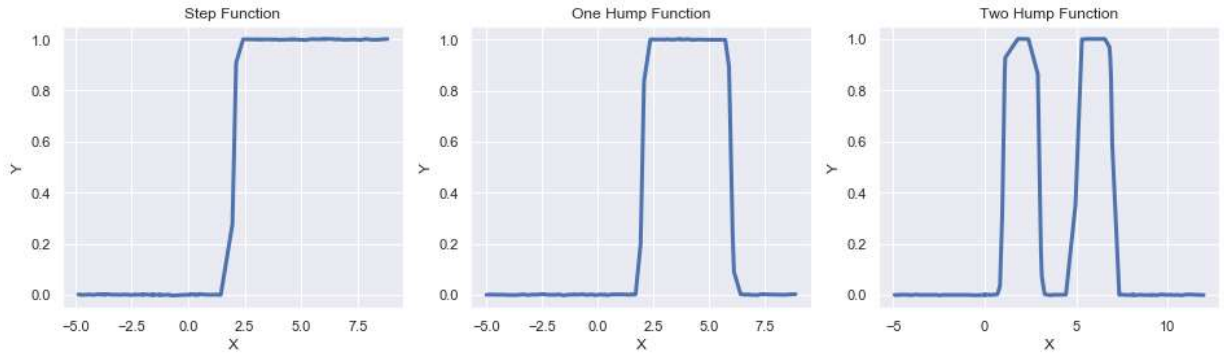
Answers

1.1

In [2463]:
```
# read in data sets as dataframes
step_df = pd.read_csv("data/step_df.csv")
one_hump_df = pd.read_csv("data/one_hump_df.csv")
two_hump_df = pd.read_csv("data/two_hump_df.csv")
```

In [2464]:
```
# plot the raw x,y data from data frame
def plot_df_xy(df, ax=None):

    # define default axes
    if ax is None:
        ax = plt.gca()

    # get permutation for sorted values of x
    x_permutation = np.argsort(df.x.values)

    # plot line in sequential increasing values of x
    ax.plot(df.x.values[x_permutation], df.y.values[x_permutation], label='raw da
    plt.xlabel("X")
    plt.ylabel("Y")
    return((ax,x_permutation))
```

In [2465]:
```
# visualize raw data for all 3 dataframes
plt.figure(figsize=(16,4))
plt.subplot(1,3,1)
plot_df_xy(step_df)
plt.title("Step Function")

plt.subplot(1,3,2)
plot_df_xy(one_hump_df)
plt.title("One Hump Function")

plt.subplot(1,3,3)
plot_df_xy(two_hump_df)
plt.title("Two Hump Function");
```



1.2 Give values to the weights manually, perform a forward pass using the data for the single step function and a hidden layer of one node, and plot the output from the network, in the same plot as the true yy values. Adjust the weigths (again manualy) until the plots match as closely as possible.

We can start by defining functions for our linear and sigmoid functions.

In [2466]:
```
# calculate z: the linear combination of the input data with the
# mode weights and biases
def z_fun(x: np.ndarray, w):
    """
    Compute z: the linear combination of the input data with the
    mode weights and biases
      Inputs:
         x: (m x n) vector of layer input data
         w: (n x p) vector of weights in the current layer
         b: (1 x p) vector of biases in the current layer
      Outputs:
         z: (num_observations x num_nodes) array of outputs from current layer
     """

    # ensure arrays are correct shape for matrix multiplication
    x = x.reshape(x.shape[0],-1)
    x = np.concatenate((x,np.ones((x.shape[0],1))),axis=1)

    z = np.matmul(x,w)
    return(z)
```

In [2467]:
```
# define sigmoid function
def sig_fun(z):
    """
    Compute sigmoid:
      Inputs:
         z: ndarray of values
      Outputs:
         h: ndarray of shape(z) containing sigmoid(z)
     """
    return(1/(1+np.exp(-z)))
```

There is no need to define a separate linear output function, as we can just repurpose `z_fun` above for the output as well. Now we can define a single function to to run a single iteration of the network.

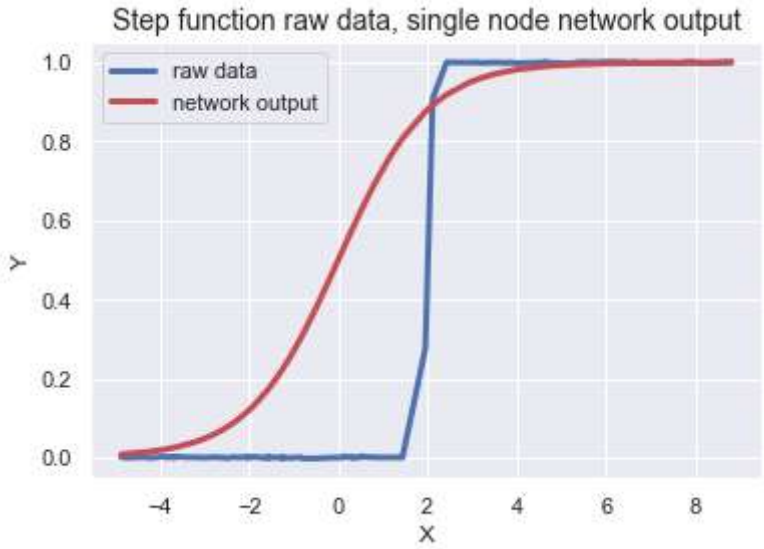
In [2468]:
```
def forward_network(x, hid_w, out_w):
    """
    Compute y: the output of single hidden layer network with
        Inputs:
            x: (num_observations x 1) vector of raw data where N is num observati
            hid_w: (1 x num_hidden) vector of weights in the network hidden layer
            hid_b: (1 x num_hidden) vector of biases in the network hidden layer
            hid_w: (1 x num_output) vector of weights in the network output layer
            hid_b: (1 x num_output) vector of biases in the network output layer
        Outputs:
            out_h: (num_observations x num_output) array of outputs from network
    """

    # input the values to layer one linear function and then activate sigmoid
    z = z_fun(x, hid_w)
    h = sig_fun(z)

    # pass hidden layer output to the output layer
    out_h = z_fun(h, out_w)
    return(out_h)
```

To evaluate the models performance, we should define a function to compare the models response var to the true response.

In [2469]:
```
# plot the input data frame and compare to the model output
def plot_model_comparison(df, model_output, label="network output"):

    ax, perm = plot_df_xy(df)
    ax.plot(df.x.values[perm], model_output[perm], label=label, lw=3)
    ax.legend()
```

Let's test out the model by initializing some parameters, running the input through the network and plotting the results

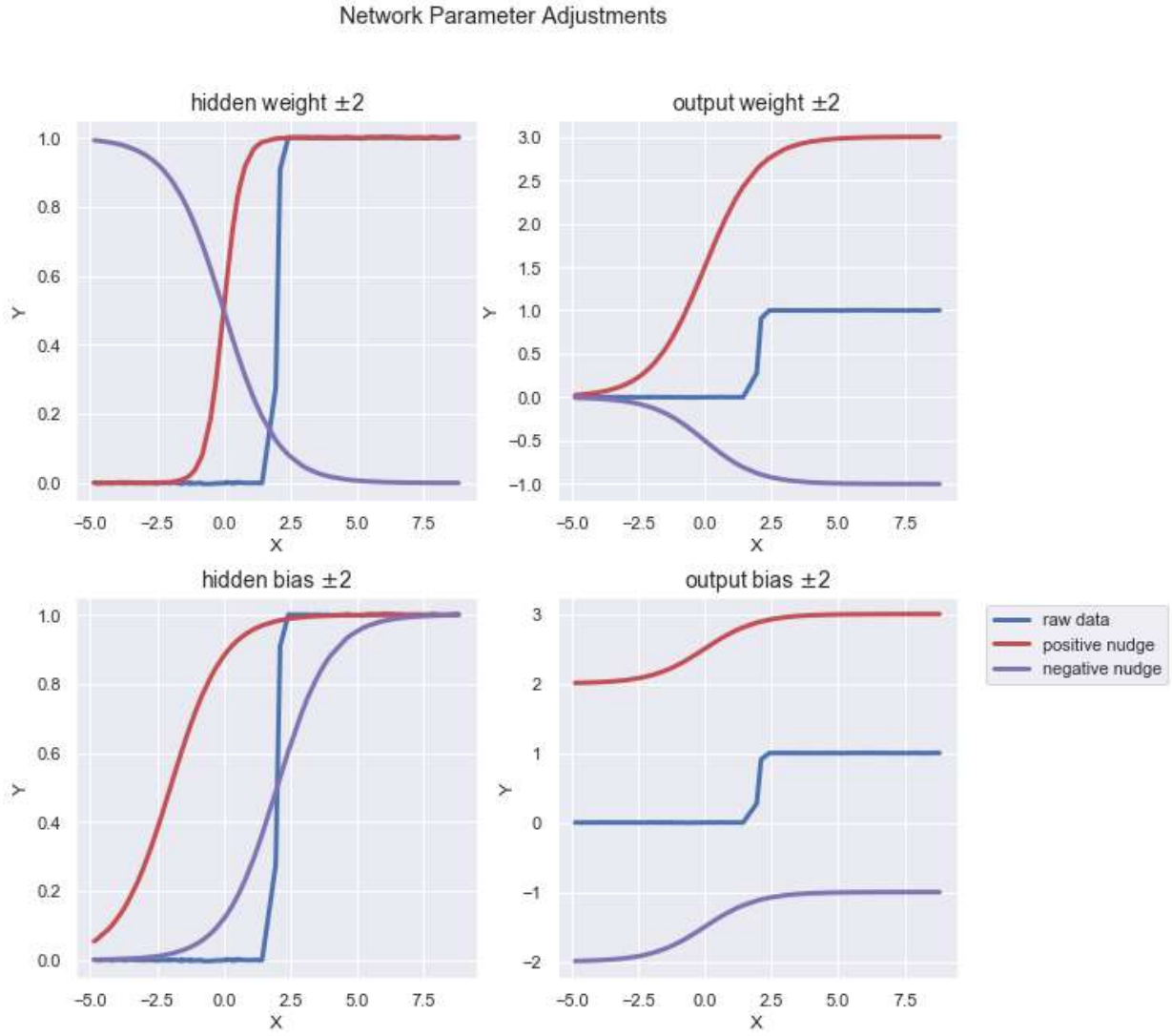
In [2470]:
```
# well start with boths weights=1 and both biases=0
x = step_df.x.values
hidden_w = np.array([1,0]).reshape(2,1)
output_w = np.array([1,0]).reshape(2,1)

output = forward_network(x, hidden_w, output_w)

# plot the output
plot_model_comparison(step_df, output)
plt.title("Step function raw data, single node network output", fontsize=14);
```



The output looks like and upside down gaussian. Let's try adjusting each of our model weights/biases independently by nudging the parameters up and down to visualize the effects of each parameter on the output.

In [2471]:
```
# define parameters and amount to shift each parameter by
params = np.hstack((hidden_w,output_w))
p_names = ["hidden weight","output weight","hidden bias","output bias"]
m = params.shape[0]
n = params.shape[1]
shift = 2

# interate over parameters, shift, and plot results
plt.figure(figsize=(10,10))
for i in range(m):
    for j in range(n):

        ct = i*m + j

        # plot raw data
        plt.subplot(params.shape[0],params.shape[1],ct+1)
        ax, perm = plot_df_xy(step_df)

        # shift parameter up and plot output
        tmp_p =params.copy()
        tmp_p[i,j] += shift
        output = forward_network(x, tmp_p[:,0], tmp_p[:,1])
        ax.plot(x[perm], output[perm], lw=3)

        # shift parameter down and plot output
        tmp_p[i,j] += shift*-2
        output = forward_network(x, tmp_p[:,0], tmp_p[:,1])
        ax.plot(x[perm], output[perm], lw=3)
        plt.title("{} $\pm${}".format(p_names[ct],shift), fontsize=14);

plt.legend(plt.gca(), labels=["raw data","positive nudge","negative nudge"], bbox
plt.suptitle("Network Parameter Adjustments")
plt.subplots_adjust(hspace=.25)
```

```
C:\Users\winsl0w\Anaconda3\lib\site-packages\matplotlib\legend.py:1364: UserWar
ning: You have mixed positional and keyword arguments, some input may be discar
ded.
  warnings.warn("You have mixed positional and keyword "
```

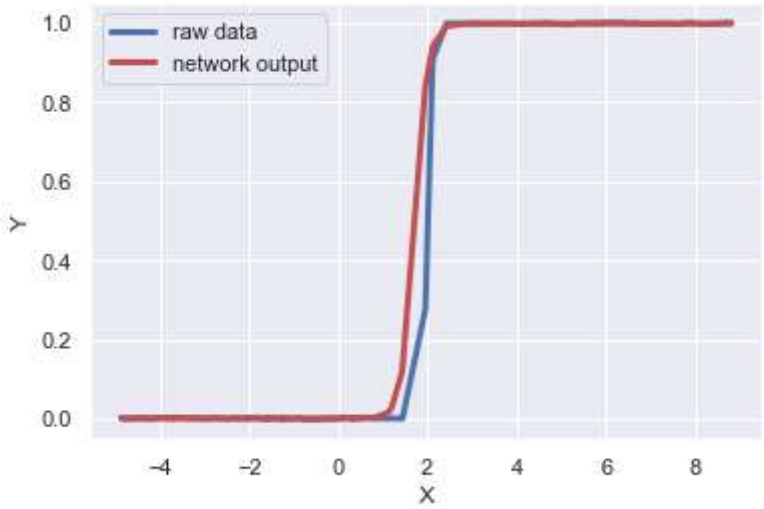


The effects of each parameter on the output might be summarized as follows:

- Hidden Weight - steepness of the step
- Hidden Bias - horizontal offset of the step
- Output Weight - amplitude of the step
- Output Bias - vertical position offset of the step

Therefore, to align the output more closely, we should make the step more steep (hidden weight) and shift it to the right (hidden_bias). By tweaking with the weights a bit in this way, we can achieve the below plot:

In [2472]:
```
hid_w = hidden_w.copy() + np.array([6,-12]).reshape(-1,1)
out_w = hidden_w.copy() + np.array([0,0]).reshape(-1,1)
output = forward_network(x, hid_w, out_w)
ax, perm = plot_df_xy(step_df)
ax.plot(x[perm], output[perm], lw=3,label="network output")
ax.legend();
```



In [2473]:
```
print("Hidden Weight/Bias: \n{}".format(hid_w))
print("Output Weight/Bias: \n{}".format(out_w))
```

```
Hidden Weight/Bias:
[[  7]
 [-12]]
Output Weight/Bias:
[[1]
 [0]]
```

This looks like a decent fit. The final weights are:

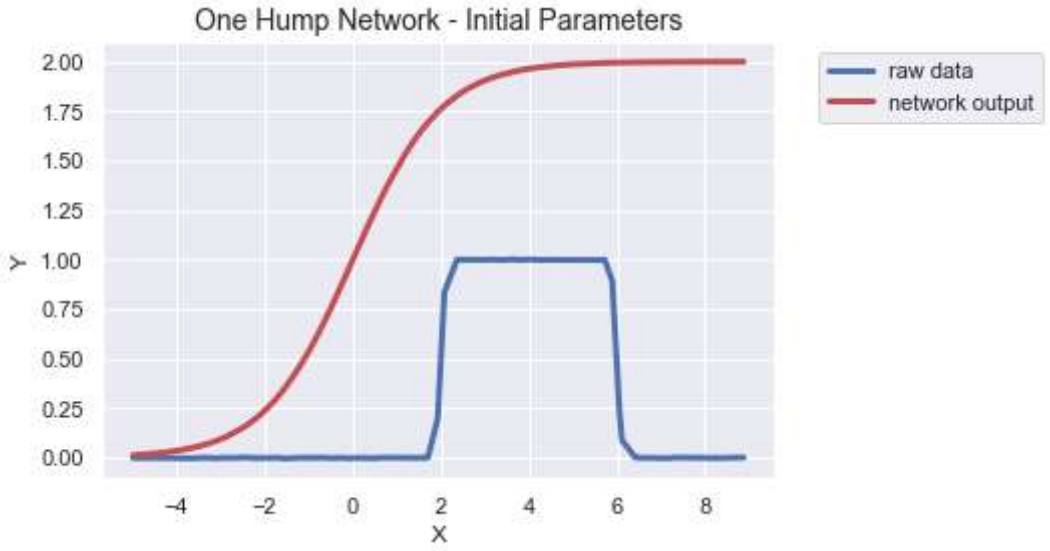
- Hidden Weight = 7
- Hidden Bias = -12
- Output Weight = 1
- Output Bias = 0

1.3 Do the same for the one hump function data, this time using a hidden layer consisting of two nodes.

Since the network function definitions above are capable of accepting additional layers, we do not need to re-define our functions, only the input weights and biases. We can start just by initializing parameters as above.

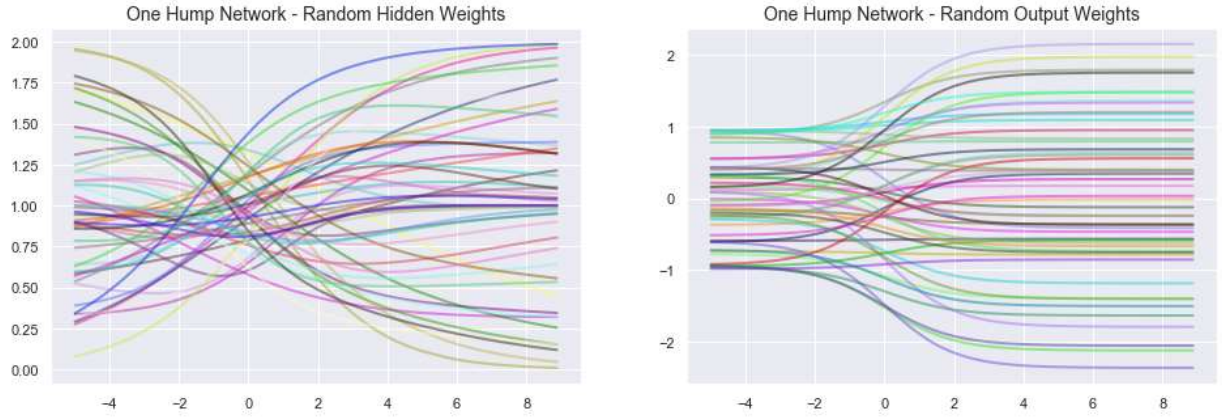
In [2474]:
```
# define the network input, weights, and biases (for hidden and output layers)
x = one_hump_df.x.values
num_nodes = 2
hidden_w = np.tile(np.array([1,0]).reshape(-1,1),(1,num_nodes))
output_w = np.append(np.array([1]*num_nodes),0).reshape(-1,1)
```

In [2475]:
```
# compute network output and plot results
output = forward_network(x, hidden_w, output_w)
ax, perm = plot_df_xy(one_hump_df)
ax.plot(x[perm], output[perm], lw=3, label="network output")
ax.legend(bbox_to_anchor=(1.05,1))
plt.title("One Hump Network - Initial Parameters", fontsize=14);
```



If we want to match the output, we should flip the output curve about the vertical axis, shift the curve to the right, broaden the width of the curve, and try to make the edges of the hump steeper. We can start investigating the effect of each parameter by shifting the hidden weights.

In [2476]:
```
# initialize some values to test
n = 50
plt.figure(figsize=(16,5))
plt.subplot(1,2,1)
for i in range(n):
    output = forward_network(x, np.random.rand(2,2)*2-1, output_w)
    plt.plot(x[perm], output[perm], lw=2, color=np.random.rand(3), alpha=0.5)
    plt.title("One Hump Network - Random Hidden Weights", fontsize=14)

plt.subplot(1,2,2)
for i in range(n):
    output = forward_network(x, hidden_w, np.random.rand(3,1)*2-1)
    plt.plot(x[perm], output[perm], lw=2, color=np.random.rand(3), alpha=0.5)
    plt.title("One Hump Network - Random Output Weights", fontsize=14)
```

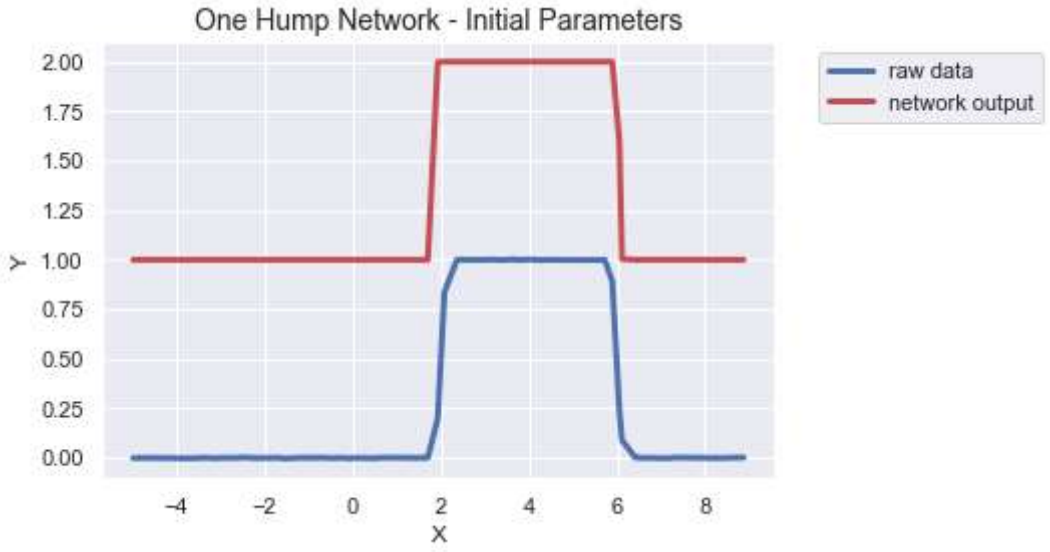


From the plot above, it looks like two sigmoids are being added together in the hidden weights plot (which makes sense given the single node model). Tuning the weights of the hidden layer seems to sort of affect the horizontal position and sign of the two sigmoid steps. When the sigmoids have opposite signs, we can see things that look like gaussians (which looks like a hump). The output weights appear to affect the amplitude and vertical position of the curve.

Referencing the strategy above:

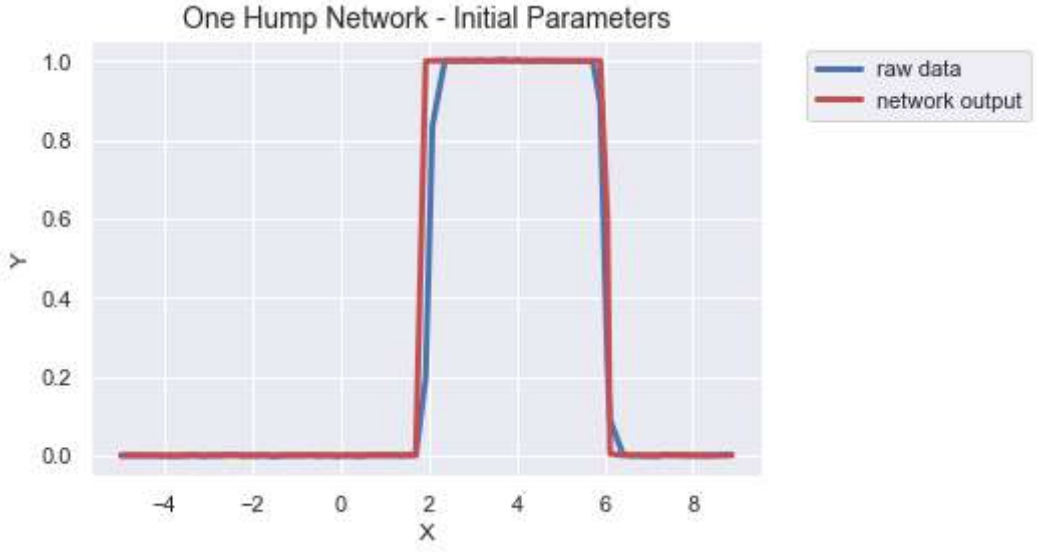
We should start by creating a bump by making the two hidden weights have opposite sign. We can then shift the bump to the left and right by changing the hidden biases. By tweaking these parameters manually we can end up with something like the plot below.

In [2477]:
```
# compute network output and plot results
hid_w = hidden_w.copy() + np.array([[80,-150],[-100,600]]).T.reshape(2,2)
out_w = output_w.copy()

output = forward_network(x, hid_w, out_w)
ax, perm = plot_df_xy(one_hump_df)
ax.plot(x[perm], output[perm], lw=3, label="network output")
ax.legend(bbox_to_anchor=(1.05,1))
plt.title("One Hump Network - Initial Parameters", fontsize=14);
```



Now we need to adjust the output bias to adjust the vertical position of the function.

In [2478]:
```
# compute network output and plot results
out_w = output_w.copy() + np.array([0,0,-1]).reshape(3,1)

output = forward_network(x, hid_w, out_w)
ax, perm = plot_df_xy(one_hump_df)
ax.plot(x[perm], output[perm], lw=3, label="network output")
ax.legend(bbox_to_anchor=(1.05,1))
plt.title("One Hump Network - Initial Parameters", fontsize=14);
```



In [2479]:
```
print("Hidden Weights/Biases:\n{}".format(hid_w))
print("Output Weights/Bias:\n{}".format(out_w))
print("MSE = %.6f" % mean_squared_error(one_hump_df.y.values,output))
```

```
Hidden Weights/Biases:
[[  81  -99]
 [-150  600]]
Output Weights/Bias:
[[ 1]
 [ 1]
 [-1]]
MSE = 0.008116
```

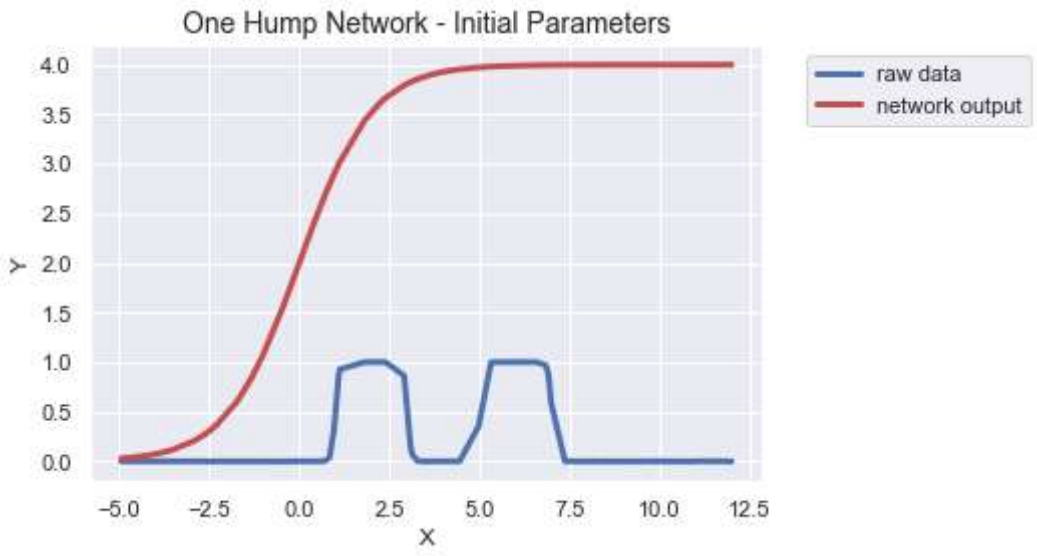
After tweaking the network manually to achieve a good fit, our parameters are as follows:

- hidden weights = (81, -99)
- hidden biases = (-150, 600)
- output weights = (1,1)
- output bias = 0

1.4 Do the same for the **two hump** function data but this time increase the number of hidden nodes to **four**.

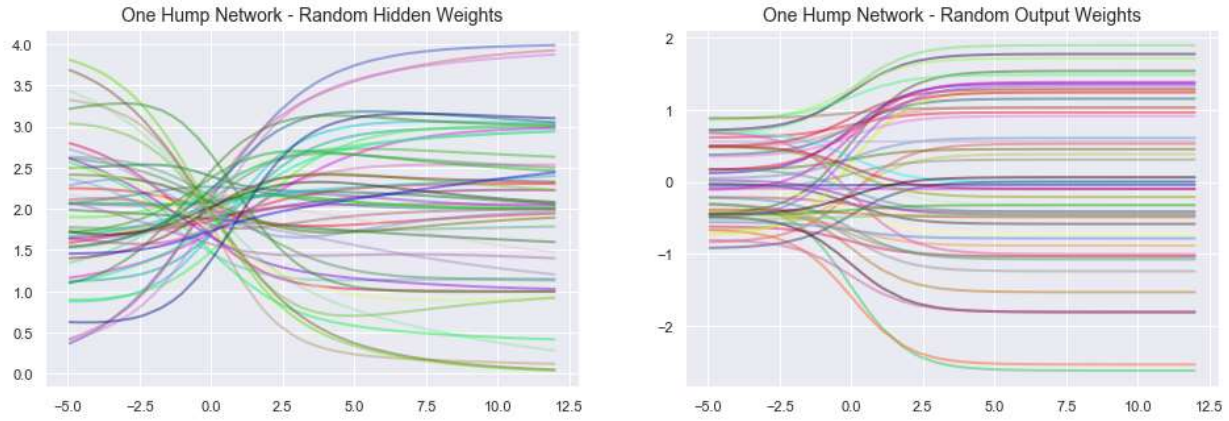
In [2480]:
```
# define the network input, weights, and biases (for hidden and output layers)
x = two_hump_df.x.values
num_nodes = 4
hidden_w = np.tile(np.array([1,0]).reshape(-1,1),(1,num_nodes))
output_w = np.append(np.array([1]*num_nodes),0).reshape(-1,1)
```

In [2481]:
```
# compute network output and plot results
output = forward_network(x, hidden_w, output_w)
ax, perm = plot_df_xy(two_hump_df)
ax.plot(x[perm], output[perm], lw=3, label="network output")
ax.legend(bbox_to_anchor=(1.05,1))
plt.title("One Hump Network - Initial Parameters", fontsize=14);
```



If we want to match the output, we should flip the output curve about the vertical axis, shift the curve to the right, broaden the width of the curve, and try to make the edges of the hump steeper. We can start investigating the effect of each parameter by shifting the hidden weights.

In [2482]:
```
# initialize some values to test
n = 50
plt.figure(figsize=(16,5))
plt.subplot(1,2,1)
for i in range(n):
    output = forward_network(x, np.random.rand(2,num_nodes)*2-1, output_w)
    plt.plot(x[perm], output[perm], lw=2, color=np.random.rand(3), alpha=0.5)
    plt.title("One Hump Network - Random Hidden Weights", fontsize=14)

plt.subplot(1,2,2)
for i in range(n):
    output = forward_network(x, hidden_w, np.random.rand(num_nodes+1,1)*2-1)
    plt.plot(x[perm], output[perm], lw=2, color=np.random.rand(3), alpha=0.5)
    plt.title("One Hump Network - Random Output Weights", fontsize=14)
```

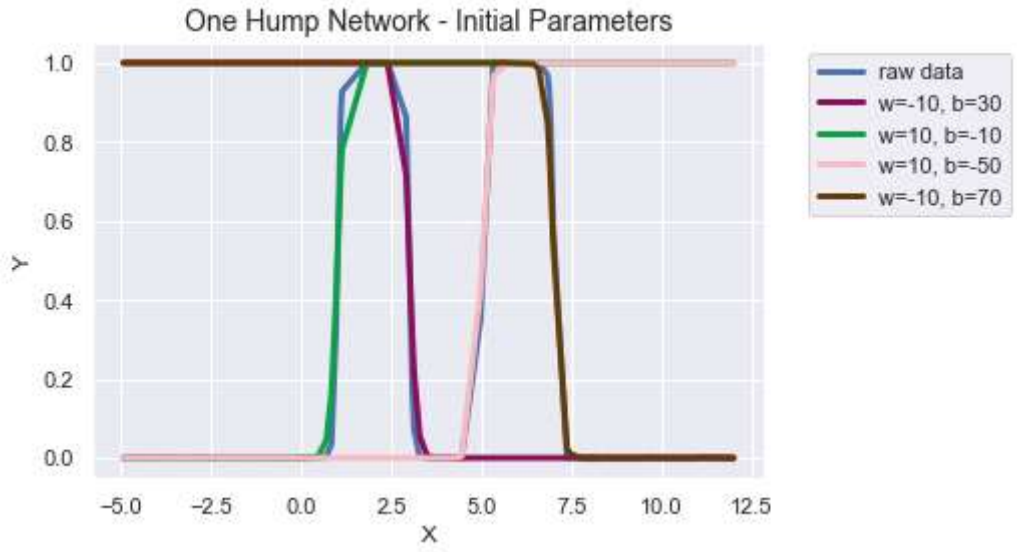


This looks similar to our two node plot from before. From the plot above and the previous problem, we can infer that four sigmoids are being added together in the hidden weights plot. As before, tuning the weights of the hidden layer seems to sort of affect the horizontal position and sign of the four sigmoid steps. When the sigmoids have opposite signs, we can see things that look like gaussians (which looks like a hump). The output weights appear to affect the amplitude and vertical position of the curve.

We'll start with the output weights. To make things simpler as we are tweaking the individual weights, we can visualize the output curve of each hidden layer individually.

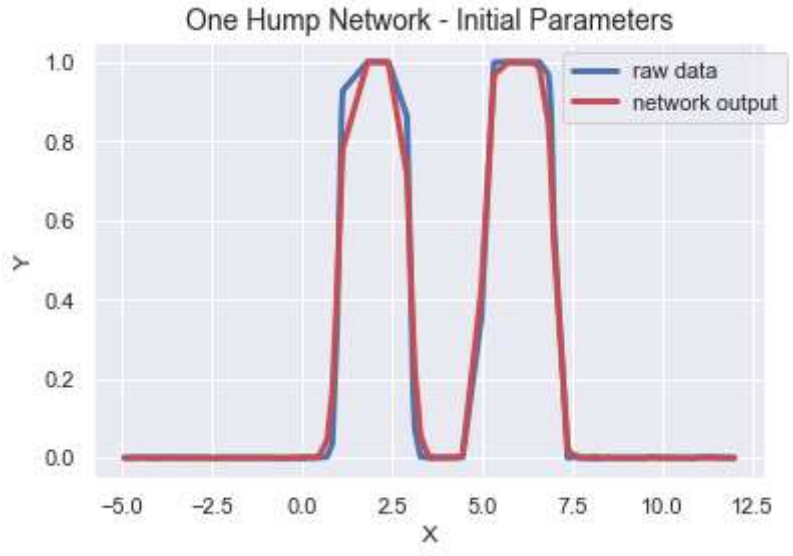
In [2483]:
```
def plot_hidden_outputs(x, w):

    z = z_fun(x,w)
    h = sig_fun(z)

    perm = np.argsort(x)
    for i in range(h.shape[1]):
        label = "w={}, b={}".format(w[0,i],w[1,i])
        plt.gca().plot(x[perm],h[perm,i],label=label, color=np.random.rand(3), lw
    plt.legend(bbox_to_anchor=(1.05,1))

    return(h)
```

In [2484]:
```
# compute network output and plot results
hid_w = np.array([[-10,30],[10,-10],[10,-50],[-10,70]]).T.reshape(2,num_nodes)
out_w = output_w.copy()

#output = forward_network(x, hid_w, out_w)
ax, perm = plot_df_xy(two_hump_df)
h = plot_hidden_outputs(x, hid_w)
ax.legend(bbox_to_anchor=(1.05,1))
plt.title("One Hump Network - Initial Parameters", fontsize=14);
```



Using the plotting function defined above, we can position the four sigmoids around the two humps from the raw data plot. Once the curves are added together, there will be a vertical offset as before, which we can tweak with the output layer bias.

In [2485]:
```
# compute network output and plot results
out_w = output_w.copy() + np.array([0,0,0,0,-2]).reshape(num_nodes+1,1)

output = forward_network(x, hid_w, out_w)
ax, perm = plot_df_xy(two_hump_df)
ax.plot(x[perm], output[perm], lw=3, label="network output")
ax.legend(bbox_to_anchor=(1.05,1))
plt.title("One Hump Network - Initial Parameters", fontsize=14);
```



In [2486]:
```
print("Hidden Weights/Biases:\n{}".format(hid_w))
print("Output Weights/Bias:\n{}".format(out_w))
print("MSE = %.6f" % mean_squared_error(two_hump_df.y.values,output))
```

```
Hidden Weights/Biases:
[[-10  10  10 -10]
 [ 30 -10 -50  70]]
Output Weights/Bias:
[[ 1]
 [ 1]
 [ 1]
 [ 1]
 [-2]]
MSE = 0.002466
```

This looks like a pretty decent manual fit. The parameters we end up with are:

- hidden weights = (-10, 10, 10, -10)
- hidden biases = (30, -10, -50, 70)
- output weights = (1,1,1,1)
- output bias = -2

1.5 Choose the appropriate loss function and calculate and report the loss from all three cases. Derive the gradient of the output layer's weights for all three cases (step, one hump and two humps). Use the weights for the hidden layers you found in the previous question and perform

gradient descent on the weights of this layer (output layer). What is the optimised weight value and loss you obtained? How many steps did you take to reach this value? What is the threshold value you used to stop?

Step Function Gradient

Let's start with the `step_fun` since it is the simplest case. To compute the gradient of each of the output layers weights, we will need to calculate the derivative of the loss function *w.r.t* each of our output weights.

Let $L(\hat{y}, y)$ be the loss function between our predictions $\hat{y}$ and the true values y. Defining our loss function as the MSE, we get:

$$L(\hat{y}, y) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

Therefore:

$$\frac{\delta L}{\delta \hat{y}} = \frac{1}{N}\sum_{i=1}^{N} 2(\hat{y}_i - y_i)$$

Since $\hat{y}$ is the output of the affine transformation of our output layer:

$$\hat{y} = W_{out1}\, h_1 + W_{out0}$$

Where W_{out1} and W_{out0} are the output weight and bias, respectively. Therefore:

$$\frac{\delta \hat{y}}{\delta W_{out1}} = h_1$$

$$\frac{\delta \hat{y}}{\delta W_{out0}} = 1$$

Where h is the input from the previous layer. We can use the chain rule and our partial derivatives to calculate the gradient with respect to the output weight and bias as follows:

$$\frac{\delta L}{\delta W_{out1}} = \frac{\delta L}{\delta \hat{y}}\frac{\delta \hat{y}}{\delta W_{out1}}$$

$$\frac{\delta L}{\delta W_{out0}} = \frac{\delta L}{\delta \hat{y}}\frac{\delta \hat{y}}{\delta W_{out0}}$$

Therefore:

$$\frac{\delta L}{\delta W_{out1}} = \frac{1}{N}\sum_{i=1}^{N} 2(\hat{y}_i - y_i)h_{1i}$$

$$\frac{\delta L}{\delta W_{out0}} = \frac{1}{N}\sum_{i=1}^{N} 2(\hat{y}_i - y_i)$$

With these derivatives, we can calculate adjusted weights W' using the learning rate λ:

$$W' = W - \lambda \frac{\delta L}{\delta W}$$

One Hump Function

To compute the gradient of output layer for the two node network we used to apporximate the `one_hump_fun`, we just need to extend the above derivation to include an additional weight in the output layer (W_{out2}) and an additional input to the output layer (h_2). Our loss function is not changing, and so $\frac{\delta L}{\delta \hat{y}}$ will not change. In the two node network:

$$\hat{y} = W_{out2} h_2 + W_{out1} h_1 + W_{out0}$$

Since W_{out1} and W_{out0} do not depend, on W_{out2} or h_2, we can use the expressions we derived above for the derivatives of those terms as well. All we need to do is calculate:

$$\frac{\delta \hat{y}}{\delta W_{out2}} = h_2$$

Using the chain rule again as above:

$$\frac{\delta L}{\delta W_{out2}} = \frac{1}{N}\sum_{i=1}^{N} 2(\hat{y}_i - y_i) h_{2i}$$

Two Hump Function

We can see now that to compute the derivative of the loss function of our four-layer network, we are just adding additional output layer weights $(W_{out3,4})$ and an additional input to the output layer $(h_3, 4)$. We can use symmetry to infer that the derivative of the loss function $L(\hat{y}, y)$ *w.r.t.* these new weights will be:

$$\frac{\delta L}{\delta W_{out3}} = \frac{1}{N}\sum_{i=1}^{N} 2(\hat{y}_i - y_i) h_{3i}$$

$$\frac{\delta L}{\delta W_{out4}} = \frac{1}{N}\sum_{i=1}^{N} 2(\hat{y}_i - y_i) h_{4i}$$

Using the above derivations, we can define a general framework for computing the output layer gradients.

In [2487]:
```
def compute_grad(y_hat, y_true, h):

    # add bias constant to previous layer output
    h_with_bias = np.hstack((h,np.ones((h.shape[0],1))))

    # compute the inside term
    y_delta = ((y_hat.flatten() - y_true.flatten())*2).reshape(-1,1)

    # multiply by h and caluclate mean
    return(np.mean((y_delta)*h_with_bias, axis=0))
```

Now we can add this feature to a new network function which does the forward output and backpropogation. We can output the updated values of the weights and iterate over it multiple times to train the net.

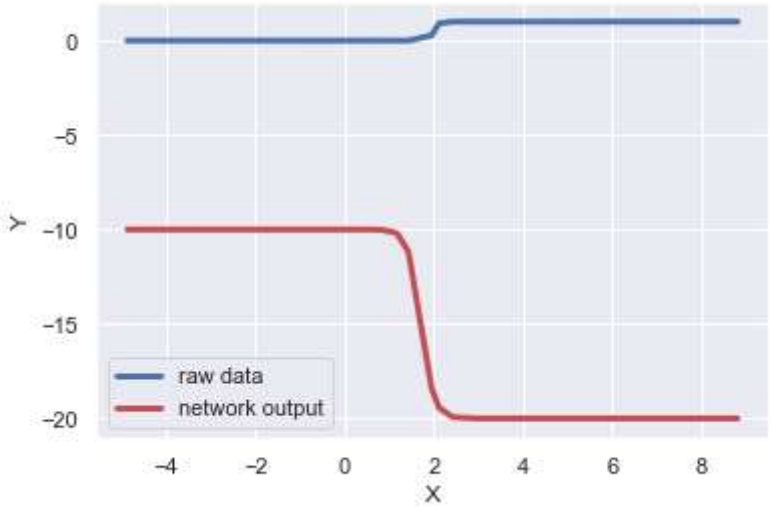
In [2488]:
```
def forward_backward_network(x, y, lambduh, hid_w, out_w):
    """
    Compute y: the output of single hidden layer network with
        Inputs:
            x: (num_observations x 1) vector of raw data where N is num observati
            hid_w: (1 x num_hidden) vector of weights in the network hidden layer
            out_w: (1 x num_output) vector of weights in the network output layer
        Outputs:
            out_h: (num_observations x num_output) array of outputs from network
    """

    # input the values to layer one linear function and then activate sigmoid
    z = z_fun(x, hid_w)
    h = sig_fun(z)

    # pass hidden layer output to the output layer
    y_hat = z_fun(h, out_w)

    # compute the loss
    e = y_hat.flatten()-y_true.flatten()
    loss = mean_squared_error(y,y_hat)

    # compute gradient for output layer
    grad = compute_grad(y_hat, y, h)

    # calculate new weights for output layer
    new_out_w = out_w - grad.reshape(out_w.shape[0],out_w.shape[1])*lambduh

    return(y_hat, new_out_w, loss)
```

Gradient Descent - Step Function

We're ready to try it out on our first function. Let's start on the `step_fun` since it's the simplest. We can define our hidden weights as determined above and intentionally assign bad output weights/bias to see if the gradient descent is working.

In [2489]:
```
# define parameters for gradient descent
reps = 1000
x = step_df.x.values
y = step_df.y.values
lamduh = .5
hid_w = np.array([7,-12]).reshape(2,1)
out_w = np.array([-10,-10]).reshape(2,1)
```

In [2490]:
```
# check plot before descent
output = forward_network(x, hid_w, out_w)
ax, perm = plot_df_xy(step_df)
ax.plot(x[perm], output[perm], lw=3,label="network output")
ax.legend();
```

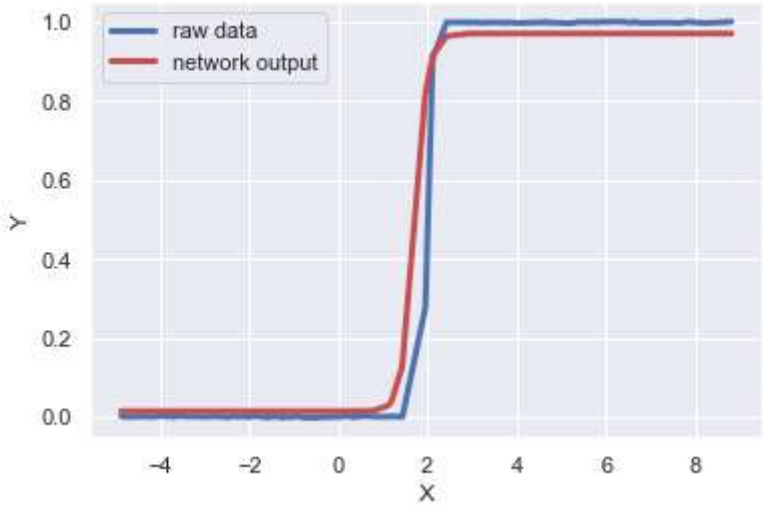


We can see we're starting out with a bad fit. Let's do some gradient descent! We'll define an low cutoff loss value = 0.004 to determine when to stop iterating. This was the loss calculated with the handpicked parameters above; so we know that it should be possible to do at least that well.

In [2491]:
```
loss = 1
loss_threshold = 0.004
i = 0
while loss > loss_threshold:
    i+=1
    y_hat, out_w, loss = forward_backward_network(x, y, lamduh, hid_w, out_w)

print("Loss = %.3f after %i iterations" % (loss,i))
print("Output weights: \n {}".format(out_w))
```

```
Loss = 0.004 after 22 iterations
Output weights:
 [[0.95806161]
 [0.01357317]]
```

In [2492]:
```
# check plot after descent
output = forward_network(x, hid_w, out_w)
ax, perm = plot_df_xy(step_df)
ax.plot(x[perm], output[perm], lw=3,label="network output")
ax.legend();
```



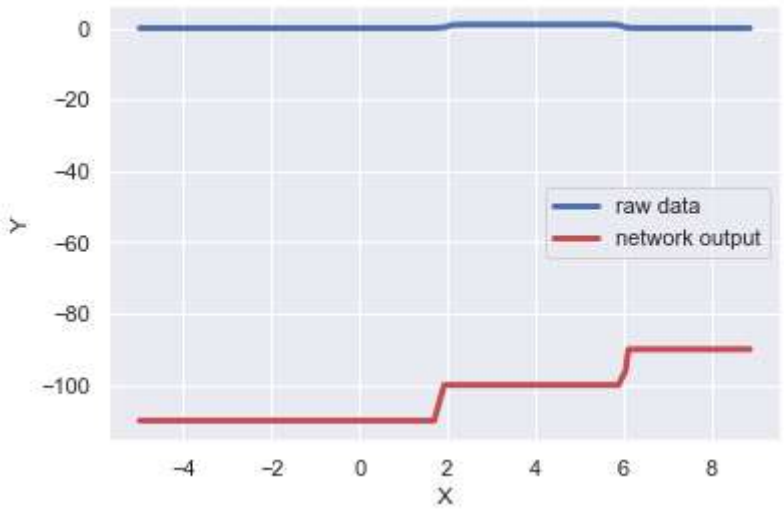
The fit definitely looks better than when we first started. We know the fit isn't likely to look as good as our manual fit unless we set a lower cutoff. It only took 22 iterations to reach the threshold set above with $\lambda = 0.5$. We could of course adjust λ and the number of iterations will most likely change.

Gradient Descent - One Hump Function

We'll define our hidden weights as determined above and intentiontally assign bad output weights/bias to see if the gradient descent is working.

In [2493]:
```
# define parameters for gradient descent
reps = 10000
x = one_hump_df.x.values
y = one_hump_df.y.values
lamduh = 0.5
hid_w = np.array([[81,-150],[-99,600]], dtype=float).T.reshape(2,2)
out_w = np.array([10,-10,-100], dtype=float).reshape(3,1)
```

In [2494]:
```
# check plot before descent
output = forward_network(x, hid_w, out_w)
ax, perm = plot_df_xy(one_hump_df)
ax.plot(x[perm], output[perm], lw=3,label="network output")
ax.legend();
```

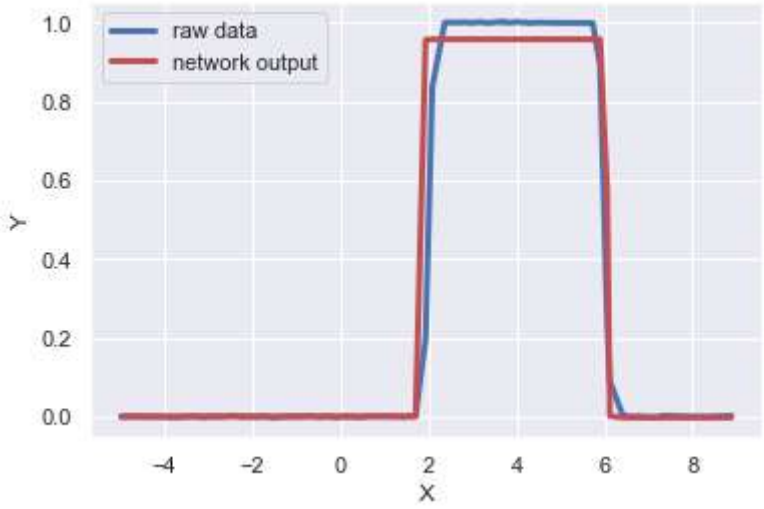


We can see we're starting out with a bad fit. Let's do some gradient descent! As before, we'll define a low cutoff loss value = 0.004 to determine when to stop iterating. This was the loss calculated with the handpicked parameters above; so we know that it should be possible to do at least that well.

In [2495]:
```
loss = 1
loss_threshold = 0.004
i = 0
while loss > loss_threshold and i < reps:
    i+=1
    y_hat, out_w, loss = forward_backward_network(x, y, lamduh, hid_w, out_w)

print("Loss = %.6f after %i iterations" % (loss,i))
print("Output weights: \n {}".format(out_w))
```

```
Loss = 0.007567 after 10000 iterations
Output weights:
 [[ 0.95750162]
 [ 0.96009282]
 [-0.96003712]]
```

In [2496]:
```
# check plot after descent
output = forward_network(x, hid_w, out_w)
ax, perm = plot_df_xy(one_hump_df)
ax.plot(x[perm], output[perm], lw=3,label="network output")
ax.legend();
```



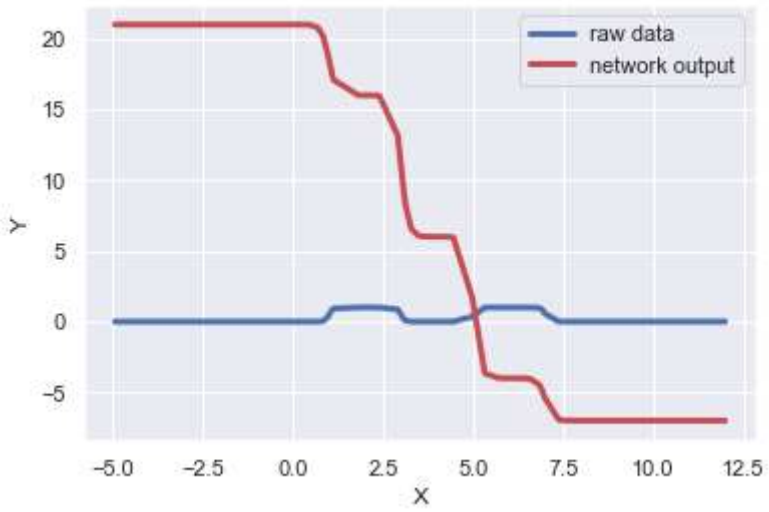
This loss is actually lower than what we got with our manual fit. This time we exceeded the maximim number of iterations (10k). This likely represents about the best we can do.

Gradient Descent - Two Hump Function

We'll define our hidden weights as determined above and intentiontally assign bad output weights/bias to see if the gradient descent is working.

In [2497]:
```
# define parameters for gradient descent
reps = 10000
x = two_hump_df.x.values
y = two_hump_df.y.values
lamduh = 0.2
hid_w = np.array([[-10,30],[10,-10],[10,-50],[-10,70]], dtype=float).T.reshape(2,
out_w = np.array([10,-5,-10,3,8], dtype=float).reshape(5,1)
```

In [2498]:
```
# check plot before descent
output = forward_network(x, hid_w, out_w)
ax, perm = plot_df_xy(two_hump_df)
ax.plot(x[perm], output[perm], lw=3,label="network output")
ax.legend();
```

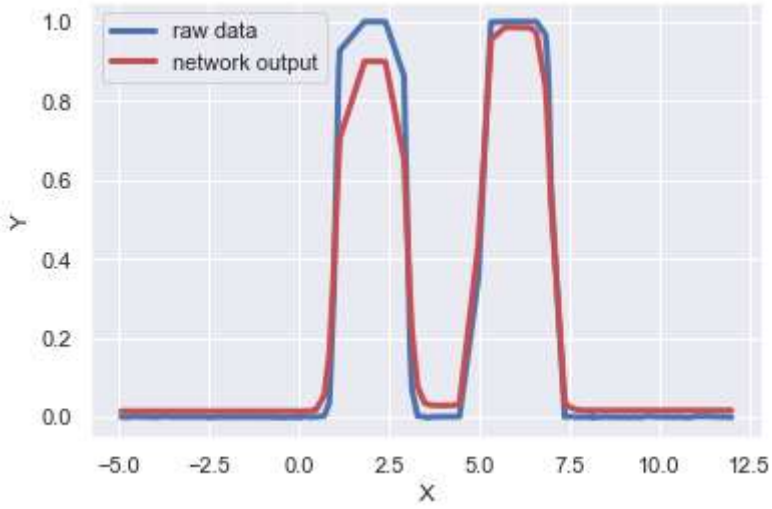


We can see we're starting out with a bad fit. Let's do some gradient descent! As before, we'll define a low cutoff loss value = 0.004 to determine when to stop iterating. This was the loss calculated with the handpicked parameters above; so we know that it should be possible to do at least that well.

In [2499]:
```
loss = 1
loss_threshold = 0.004
i = 0
while loss > loss_threshold and i < reps:
    i+=1
    y_hat, out_w, loss = forward_backward_network(x, y, lamduh, hid_w, out_w)

print("Loss = %.6f after %i iterations" % (loss,i))
print("Output weights: \n {}".format(out_w))
```

```
Loss = 0.003984 after 660 iterations
Output weights:
 [[ 0.87143474]
 [ 0.88595236]
 [ 0.9582091 ]
 [ 0.97049824]
 [-1.82772818]]
```

In [2500]:
```
# check plot after descent
output = forward_network(x, hid_w, out_w)
ax, perm = plot_df_xy(two_hump_df)
ax.plot(x[perm], output[perm], lw=3,label="network output")
ax.legend();
```



The loss here is pretty comparable to what we saw with our manual fit (a bit worse). This time it took 660 iterations, considerably less than the maximum and considerably more than the `step_fun` training time.

Question 2: Working with missing data. [50 pts]

In this exercise we are going to use the **Pima Indians onset of diabetes** dataset found in `pima-indians-diabetes.csv` . This dataset describes patient medical record data for Pima Indians and whether they had an onset of diabetes within five years. It is a binary classification problem (onset of diabetes as 1 or not as 0). The input variables that describe each patient are numerical and have varying scales. The list below shows the eight attributes plus the target variable for the dataset:

- Number of times pregnant.
- Plasma glucose concentration a 2 hours in an oral glucose tolerance test.
- Diastolic blood pressure (mm Hg).
- Triceps skin fold thickness (mm).
- 2-Hour serum insulin (mu U/ml).
- Body mass index.
- Diabetes pedigree function.
- Age (years).
- **Outcome** (1 for early onset of diabetes within five years, 0 for not), target class.

2.1. Load the dataset into a pandas dataframe named `pima_df` . Clean the data by looking at the various features and making sure that their values make sense. Look for missing data including disguised missing data. The problem of disguised missing data arises when missing data values are not explicitly represented as such, but are coded with values that can be misinterpreted as valid data. Comment on your findings.

2.2 Split the dataset into a 75-25 train-test split (use `random_state=9001`). Fit a logistic regression classifier to the training set and report the accuracy of the classifier on the test set. You should use L_2 regularization in logistic regression, with the regularization parameter tuned using cross-validation (`LogisticRegressionCV`). Report the overall classification rate.

2.3 Restart with a fresh copy of the whole dataset and impute the missing data via mean imputation. Split the data 75-25 (use `random_state=9001`) and fit a regularized logistic regression model. Report the overall classification rate.

2.4 Again restart with a fresh copy of the whole dataset and impute the missing data via a model-based imputation method. Once again split the data 75-25 (same `random_state=9001`) and fit a regularized logistic regression model. Report the overall classification rate.

2.5 Compare the results in the 3 previous parts of this problem. Prepare a paragraph (5-6 sentences) discussing the results, the computational complexity of the methods, and explain why you get the results that you see.

2.6 This question does not have one answer and requires some experimentation. Check which coefficients changed the most between the model in 2.1-2.2 and the models in 2.3 and 2.4. Are they the coefficients you expected to change given the imputation you performed? If not explain why (supporting your explanation using the data is always a good idea).

Answers

2.1 Load the dataset into a pandas dataframe named pima_df. Clean the data by looking at the various features and making sure that their values make sense. Look for missing data including disguised missing data. The problem of disguised missing data arises when missing data values are not explicitly represented as such, but are coded with values that can be misinterpreted as valid data. Comment on your findings.

In [2501]:
```
# read in data and check dtypes
pima_df = pd.read_csv("data/pima-indians-diabetes.csv")
pima_df.dtypes
```

Out[2501]:
```
Pregnancies                 int64
Glucose                     int64
BloodPressure               int64
SkinThickness               int64
Insulin                     int64
BMI                       float64
DiabetesPedigreeFunction  float64
Age                         int64
Outcome                    object
dtype: object
```

Immediately we can see that the response variable `Outcome` has dtype `object`. This is a bit suspicious since it should be a binary outcome. Let's look at the unique values.

In [2502]:
```
# return unique values of Outcome
pima_df.Outcome.unique()
```

Out[2502]:
```
array(['1', '0', '0\\', '1\\', '0}'], dtype=object)
```

We can see that the values themselves are strings, and that some of the values are have other characters appended to the number such as `0\\` and `0}`. We can quickly look at some summary statistics for the numeric data with `pd.describe`. To see if anything seems out of place.

In [2503]:
```
pima_df.describe()
```

Out[2503]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPe
count	764.000000	764.000000	764.000000	764.000000	764.000000	764.000000	
mean	3.853403	120.922775	69.111257	20.537958	80.070681	31.998429	
std	3.374327	32.039835	19.403339	15.970234	115.431087	7.899591	
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	
50%	3.000000	117.000000	72.000000	23.000000	34.000000	32.000000	
75%	6.000000	141.000000	80.000000	32.000000	128.250000	36.600000	
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	

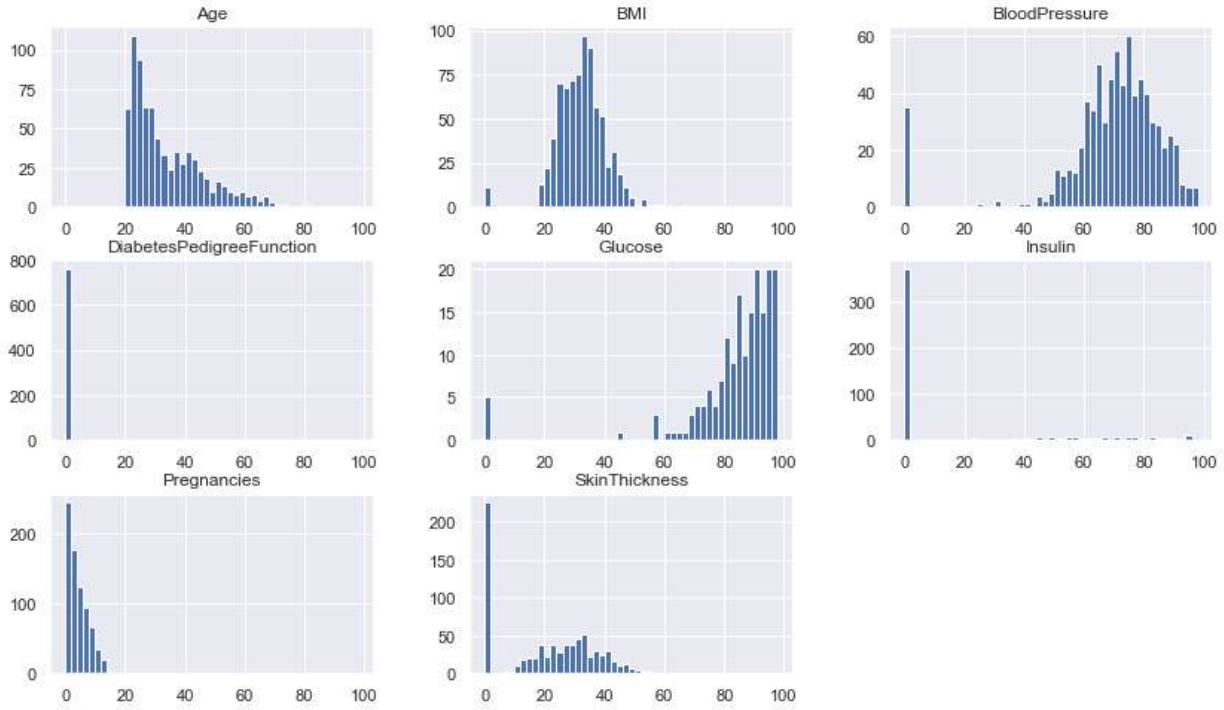
We can see that that many of our measures have a minimum of zero, which might make sense for some measures (eg pregnancies), but doesn't really make sense for some of our continuous measures such as: glucose, blood pressure, insulin, BMI. We'll investigate these values soon, but first let's get a sense of whether or not any of our predictors contain `NaN`.

In [2504]:
```
# apply isnull to each column and sum to count number of missing values
pima_df.apply(lambda x: x.isnull()).sum()
```

Out[2504]:
```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

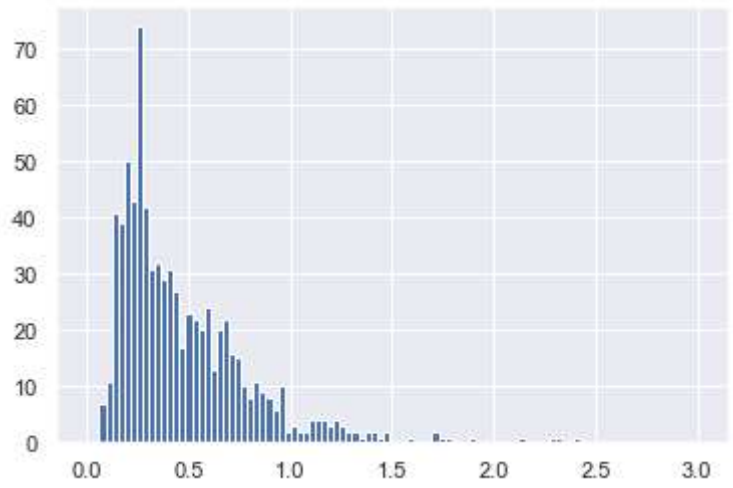
It looks like missing values for our numeric predictors are primarily `0`. We can use histograms to get a sense of whether or not zero falls within the bulk of the distribution.

In [2505]:
```
# plot histograms
pima_df.hist(figsize=(14,8), bins=np.arange(0,100,2));
```



These plots largely confirm the suspicion above. The number of pregnancies on the front tail of the distribution does seem higher than what we might expect but not outside the realm of possibility. I'm going to assume that data is accurate. The diabetes pedigree function seems to fall within the range around zero. We can plot it separately with smaller bins to inspect the data more closely.

In [2506]:
```
# plot Diabetes Pedigree separately
plt.hist(pima_df.DiabetesPedigreeFunction, bins=np.linspace(0,3,100));
```



Now we can start to clean the data. We'll start by removing the unsual characters from `Outcome` and converting it to `bool` dtype. We can recover this data because it seems clear that the presence of these extra characters is the result of some sort of error during data entry.

In [2507]:
```
# convert to boolean by checking for presence of '1'
pima_df["Outcome"] = pima_df.Outcome.apply(lambda x: '1' in x)
```

Referencing the histograms above, we can discard any row with a `0` value in `BMI`, `BloodPressure`, `Glucose`, `Insulin`, and `SkinThickness` to clean the data (although this will likely result in throwing out a lot of the data). We'll write a function to do this since it will come in handy later when we're doing model based imputing.

In [2508]:
```
def drop_missing_rows(df, missing_val=np.nan, columns=None):

    # define default columns
    if columns is None:
        columns = df.columns

    # find missing values
    if np.isnan(missing_val):
        is_missing = df[columns].isna()
    else:
        is_missing = df[columns].apply(lambda x: x==missing_val)

    # conver to a row mask by looking for missing value in any row
    is_missing = is_missing.values.any(axis=1)

    # return df with rows dropped
    return(df.drop(index=df.loc[is_missing].index))
```

In [2509]:
```
# define columns to drop zero values
replace_cols = ["BMI","BloodPressure","Glucose","Insulin","SkinThickness"]
pima_df = drop_missing_rows(pima_df, missing_val=0, columns=replace_cols)
```

In [2510]:
```
# check the size of the remaining data set
pima_df.shape[0]
```

Out[2510]:
```
391
```

Now we can inspect the dataframe header to see if it looks reasonable.

In [2511]:
```
pima_df.head(10)
```

Out[2511]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction
3	1	89	66	23	94	28.1	0.167
4	0	137	40	35	168	43.1	2.288
6	3	78	50	32	88	31.0	0.248
8	2	197	70	45	543	30.5	0.158
13	1	189	60	23	846	30.1	0.398
14	5	166	72	19	175	25.8	0.587
16	0	118	84	47	230	45.8	0.551
18	1	103	30	38	83	43.3	0.183
19	1	115	70	30	96	34.6	0.529
20	3	126	88	41	235	39.3	0.704

2.2 Split the dataset into a 75-25 train-test split (use `random_state=9001`). Fit a logistic regression classifier to the training set and report the accuracy of the classifier on the test set. You should use L_2 regularization in logistic regression, with the regularization parameter tuned using cross-validation (`LogisticRegressionCV`). Report the overall classification rate.

In [2512]:
```
# split data into train and test, sort into predictors/response
train_df, test_df = train_test_split(pima_df, test_size=0.25, random_state=9001)
x_train = train_df.drop(columns="Outcome", axis=1).values
y_train = train_df.Outcome.values
x_test = test_df.drop(columns="Outcome", axis=1).values
y_test = test_df.Outcome.values

# initialize logit model and fit to training data (newton-cg uses L2)
discard_clf = LogisticRegressionCV(Cs=20, cv=10, random_state=0, solver="newton-c
print("Best C = {0:.3f}".format(clf.C_[0]))

# report test classification accuracy
print("Train classification accuracy = {0:.3f}".format(discard_clf.score(x_train,
print("Test classification accuracy = {0:.3f}".format(discard_clf.score(x_test,y_
```

```
Best C = 11.288
Train classification accuracy = 0.823
Test classification accuracy = 0.704
```

2.3 Restart with a fresh copy of the whole dataset and impute the missing data via mean imputation. Split the data 75-25 (use `random_state=9001`) and fit a regularized logistic regression model. Report the overall classification rate.

In [2513]:
```
# read in data and check dtypes
pima_df = pd.read_csv("data/pima-indians-diabetes.csv")
```

We can now repeat the above process of cleaning the data, with the exception that this time we will replace the missing numeric data with `NaN` rather than removing the rows. Replacing the missing values with `NaN` will allow us to calculate the mean without the missing values using numpy's `nanmean` function.

In [2514]:
```
# convert to boolean by checking for presence of '1'
pima_df["Outcome"] = pima_df.Outcome.apply(lambda x: '1' in x)
pima_df["Outcome"] = pima_df.Outcome.values.astype('int')

# define columns to replace zero values and find indices of missing vals in each
is_missing = pima_df[replace_cols].apply(lambda x: x==0)

#index each column and replace with nan
for i,col in enumerate(replace_cols):
    pima_df.loc[is_missing.values[:,i],col] = np.nan
```

In [2515]:
```
# compute mean of each numeric column with missing values
col_means = pima_df[replace_cols].apply(np.nanmean)

#index each column and replace with column mean
for i,col in enumerate(replace_cols):
    pima_df.loc[is_missing.values[:,i],col] = col_means[i]
```

Quickly inspecting our work in the header, we can see that values in `Insulin` and `SkinThickness` have been replaced with the mean.

In [2516]:
```
pima_df.head(10)
```

Out[2516]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFu
0	6	148.0	72.000000	35.000000	155.659033	33.60000	
1	1	85.0	66.000000	29.000000	155.659033	26.60000	
2	8	183.0	64.000000	29.165428	155.659033	23.30000	
3	1	89.0	66.000000	23.000000	94.000000	28.10000	
4	0	137.0	40.000000	35.000000	168.000000	43.10000	
5	5	116.0	74.000000	29.165428	155.659033	25.60000	
6	3	78.0	50.000000	32.000000	88.000000	31.00000	
7	10	115.0	72.429355	29.165428	155.659033	35.30000	
8	2	197.0	70.000000	45.000000	543.000000	30.50000	
9	8	125.0	96.000000	29.165428	155.659033	32.46587	

Now we can repeat the model training process from **Q2.2**

In [2517]:
```
# split data into train and test, sort into predictors/response
train_df, test_df = train_test_split(pima_df, test_size=0.25, random_state=9001)
x_train = train_df.drop(columns="Outcome", axis=1).values
y_train = train_df.Outcome.values
x_test = test_df.drop(columns="Outcome", axis=1).values
y_test = test_df.Outcome.values

# initialize logit model and fit to training data (newton-cg uses L2)
mean_impute_clf = LogisticRegressionCV(Cs=20, cv=10, random_state=0, solver="newt
print("Best C = {0:.3f}".format(clf.C_[0]))

# report test classification accuracy
print("Train classification accuracy = {0:.3f}".format(mean_impute_clf.score(x_tr
print("Test classification accuracy = {0:.3f}".format(mean_impute_clf.score(x_tes
```

```
Best C = 11.288
Train classification accuracy = 0.773
Test classification accuracy = 0.759
```

Our cross-validated C parameter has increased slightly and the test accuracy has increased $\approx$ 6%

2.4 Again restart with a fresh copy of the whole dataset and impute the missing data via a model-based imputation method. Once again split the data 75-25 (same `random_state=9001`) and fit a regularized logistic regression model. Report the overall classification rate.

We can start by defining a function that will iterate through each column with missing values and fit an OLS model to infer missing values. We'll additionally attempt to model our uncertainty in these models by drawing error from the observed distribution.

In [2518]:
```
# impute columns (cols) of dataframe (df) via OLS model with uncertainty
def OLS_impute(df, fill_columns=None, pred_columns=None):

    # set default fill and predict columns
    if fill_columns is None:
        fill_columns = df.columns
    if pred_columns is None:
        pred_columns = df.columns

    # columns to use as predictors
    col_set = set(pred_columns)

    # record number of rows with missing values
    # stop iterating over columns if num missing stops
    # decreasing
    curr_ct = curr_ct = df.isna().values.sum().sum()
    prev_ct = curr_ct + 1

    while prev_ct > curr_ct:

        prev_ct = curr_ct.copy()

        for col in fill_columns:

            # grab temporary dataframe without missing vals
            tmp_df = drop_missing_rows(df)

            # fit OLS model for target col using other columns as predictors
            pred_cols = list(col_set - set([col]))
            y = tmp_df[col].values
            x = sm.add_constant(tmp_df[pred_cols].values)
            ols_mdl = OLS(y, x).fit()

            # estimate standard deviation of predicted values
            sig = np.std(ols_mdl.predict(x).flatten()-y.flatten())

            # find rows where target is missing but predictors are complete
            target_rows = df[col].isna() & ~df[pred_cols].isna().values.any(axis=

            if any(target_rows):
                # generate predictions for values to replace and add randomly sam
                err = np.random.normal(scale=sig, size=np.sum(target_rows))
                target_x = sm.add_constant(df.loc[target_rows,pred_cols].values,
                target_y = ols_mdl.predict(target_x)
                df.loc[target_rows,col] = target_y + err

            curr_ct = df.isna().values.sum().sum()

    print("Imputation Complete: {} missing values remaining".format(curr_ct))
    return(df)
```

In [2519]:
```
# re-initialize the data set
pima_df = pd.read_csv("data/pima-indians-diabetes.csv")
```

In [2520]:
```
# convert to boolean by checking for presence of '1'
pima_df["Outcome"] = pima_df.Outcome.apply(lambda x: '1' in x)
pima_df["Outcome"] = pima_df.Outcome.values.astype('int')

# define columns to replace zero values and find indices of missing vals in each
is_missing = pima_df[replace_cols].apply(lambda x: x==0)

#index each column and replace with nan
for i,col in enumerate(replace_cols):
    pima_df.loc[is_missing.values[:,i],col] = np.nan
```

Let's start by making an initial pass at filling the data with all columns. We can tell from the header than some rows will contain multiple missing vals, which means we will not be able to predict for those rows with all predictors.

In [2521]:
```
# make an initial pass at filling the data with all columns
pima_df = OLS_impute(pima_df)
print("Num rows with missing values = {}".format(pima_df.isna().any(axis=1).sum()))
print("\nNum missing in each column:")
pima_df.isna().sum()
```

```
Imputation Complete: 508 missing values remaining
Num rows with missing values = 233

Num missing in each column:
```

Out[2521]:
```
Pregnancies                 0
Glucose                     4
BloodPressure              35
SkinThickness             226
Insulin                   233
BMI                        10
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

Looking at the number of missing values in each column, it looks like the likely culprits are `SkinThickness` and `Insulin`. So let's start by attempting to replace the missage values in `Glucose`, `BloodPressure` and `BMI` using the columns with no missing values. Then we'll replace all remaining values using only the columns with no missing values as predictors.

In [2522]:
```
# impute only BMI and glucose using complete columns only to model
pred_cols = pima_df.columns.values[~pima_df.isna().any(axis=0)]
pima_df = OLS_impute(pima_df, fill_columns=["BMI","Glucose","BloodPressure"], pre
print("Num rows with missing values = {}".format(pima_df.isna().any(axis=1).sum()
print("\nNum missing in each column:")
pima_df.isna().sum()
```

```
Imputation Complete: 459 missing values remaining
Num rows with missing values = 233

Num missing in each column:
```

Out[2522]:
```
Pregnancies                   0
Glucose                       0
BloodPressure                 0
SkinThickness               226
Insulin                     233
BMI                           0
DiabetesPedigreeFunction      0
Age                           0
Outcome                       0
dtype: int64
```

Now we'll impute the last two remaining columns with the rest of the data set.

In [2523]:
```
# impute only BMI and glucose using complete columns only to model
pred_cols = pima_df.columns.values[~pima_df.isna().any(axis=0)]
pima_df = OLS_impute(pima_df, fill_columns=["Insulin","SkinThickness"], pred_colu
print("Num rows with missing values = {}".format(pima_df.isna().any(axis=1).sum()
print("\nNum missing in each column:")
pima_df.isna().sum()
```

```
Imputation Complete: 0 missing values remaining
Num rows with missing values = 0

Num missing in each column:
```

Out[2523]:
```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

Now that the data set is filled, we can train the logistic regression model as before.

In [2524]:
```
# split data into train and test, sort into predictors/response
train_df, test_df = train_test_split(pima_df, test_size=0.25, random_state=9001)
x_train = train_df.drop(columns="Outcome", axis=1).values
y_train = train_df.Outcome.values
x_test = test_df.drop(columns="Outcome", axis=1).values
y_test = test_df.Outcome.values

# initialize logit model and fit to training data (newton-cg uses L2)
model_impute_clf = LogisticRegressionCV(Cs=20, cv=10, random_state=0, solver="new
print("Best C = {0:.3f}".format(clf.C_[0]))

# report test classification accuracy
print("Train classification accuracy = {0:.3f}".format(model_impute_clf.score(x_t
print("Test classification accuracy = {0:.3f}".format(model_impute_clf.score(x_te
```

```
Best C = 11.288
Train classification accuracy = 0.780
Test classification accuracy = 0.785
```

2.5 Compare the results in the 3 previous parts of this problem. Prepare a paragraph (5-6 sentences) discussing the results, the computational complexity of the methods, and explain why you get the results that you see.

Of the 3 methods for handling missing data, model imputation produced the best test accuracy (model_imput=0.77 compared to 0.70 and 0.76 for discarding and mean imputation respectively). Both model imputation and mean imputation produced notably higher test accuracy than discarding rows with missing data. The large difference between train and test accuracy (0.82 and 0.70 respectively) following discarding missing value rows set suggests that this improvement in test accuracy following model imputation and mean imputation primarily reflects a reduction in overfitting due to increased size of the training data set. The computational complexity of both row discarding and mean imputation should be trivial for most data sets (though mean imputation should be slightly slower). Model imputation is the most complex and will scale exponentially as the number of predictors increases. Not only do we have to fit a model on each individual predictor with missing values (possibly multiple times), but each individual model requires coefficient estimates for each other predictor.

2.6 This question does not have one answer and requires some experimentation. Check which coefficients changed the most between the model in 2.1-2.2 and the models in 2.3 and 2.4. Are they the coefficients you expected to change given the imputation you performed? If not explain why (supporting your explanation using the data is always a good idea).

In [2525]:
```
# read in data and check dtypes, check number of missing values in each column
pima_df = pd.read_csv("data/pima-indians-diabetes.csv")
print("\nNum missing in each column:")
pima_df.apply(lambda x: x==0).sum()
```

```
Num missing in each column:
```

Out[2525]:
```
Pregnancies                 111
Glucose                       5
BloodPressure                35
SkinThickness               226
Insulin                     371
BMI                          11
DiabetesPedigreeFunction      0
Age                           0
Outcome                       0
dtype: int64
```

Note: `Pregnancies` is not actually missing data but the method above is not sensitive to that.

In [2526]:
```
# compare mean impute to discard
diff = mean_impute_clf.coef_[0] - discard_clf.coef_[0]
for i,col in enumerate(pima_df.drop("Outcome",axis=1)):
    print("Before=%.3f, After=%.3f, Change=%.3f, Column=%s" % \
          (discard_clf.coef_[0][i],mean_impute_clf.coef_[0][i],diff[i],col))
```

```
Before=0.042, After=0.153, Change=0.111, Column=Pregnancies
Before=0.045, After=0.039, Change=-0.006, Column=Glucose
Before=0.006, After=-0.010, Change=-0.016, Column=BloodPressure
Before=-0.001, After=-0.008, Change=-0.007, Column=SkinThickness
Before=-0.002, After=-0.001, Change=0.001, Column=Insulin
Before=0.083, After=0.105, Change=0.022, Column=BMI
Before=0.836, After=0.984, Change=0.149, Column=DiabetesPedigreeFunction
Before=0.032, After=0.015, Change=-0.017, Column=Age
```

In [2527]:
```
# compare model impute to discard
diff =  model_impute_clf.coef_[0] - discard_clf.coef_[0]
for i,col in enumerate(pima_df.drop("Outcome",axis=1)):
    print("Before=%.3f, After=%.3f, Change=%.3f, Column=%s" % \
          (discard_clf.coef_[0][i],model_impute_clf.coef_[0][i],diff[i],col))
```

```
Before=0.042, After=0.151, Change=0.109, Column=Pregnancies
Before=0.045, After=0.037, Change=-0.008, Column=Glucose
Before=0.006, After=-0.011, Change=-0.017, Column=BloodPressure
Before=-0.001, After=-0.005, Change=-0.003, Column=SkinThickness
Before=-0.002, After=0.001, Change=0.003, Column=Insulin
Before=0.083, After=0.104, Change=0.021, Column=BMI
Before=0.836, After=0.849, Change=0.013, Column=DiabetesPedigreeFunction
Before=0.032, After=0.015, Change=-0.017, Column=Age
```

The two predictors with the largest change were `Pregnancies` and `DiabetesPedigreeFunction`, which is might seem surprising given that they actually were not missing any values. It is worth noting that these are two of the strongest predictors in each model.

In both cases, the magnitude of these predictors decreases after imputing. One possibility is that adding additional noise to the other predictors via imputation increased the relative importance of these other predictors. The same logic could be applied to having a larger sample size. The variance in our parameter estimates should decrease with larger sample size, meaning that we are less likely to get artificially large coefficients due to sampling error. Our value of C is relatively low for each of these models (11.3), meaning that these coefficients are constrained. Better sampling on `Pregnancies` and `DiabetesPedigreeFunction` likely makes it easier to pick up on the real signal in these parameters over the noise in the others.