CS109A Introduction to Data Science

Homework 6: Multilayer Feedforward Network - Dealing with Missing Data

Harvard University
Fall 2018
Instructors: Pavlos Protopapas, Kevin Rader

In [1]:
```
#RUN THIS CELL
import requests
from IPython.core.display import HTML
styles = requests.get("https://raw.githubusercontent.com/Harvard-IACS/2018-CS109
HTML(styles)
```
Out[1]:

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 [2]:
```
%matplotlib inline
import numpy as np
import numpy.random as nd
import pandas as pd
import math
import matplotlib.pyplot as plt

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

from sklearn.linear_model import LogisticRegressionCV
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
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.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from IPython.display import display
```

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**.

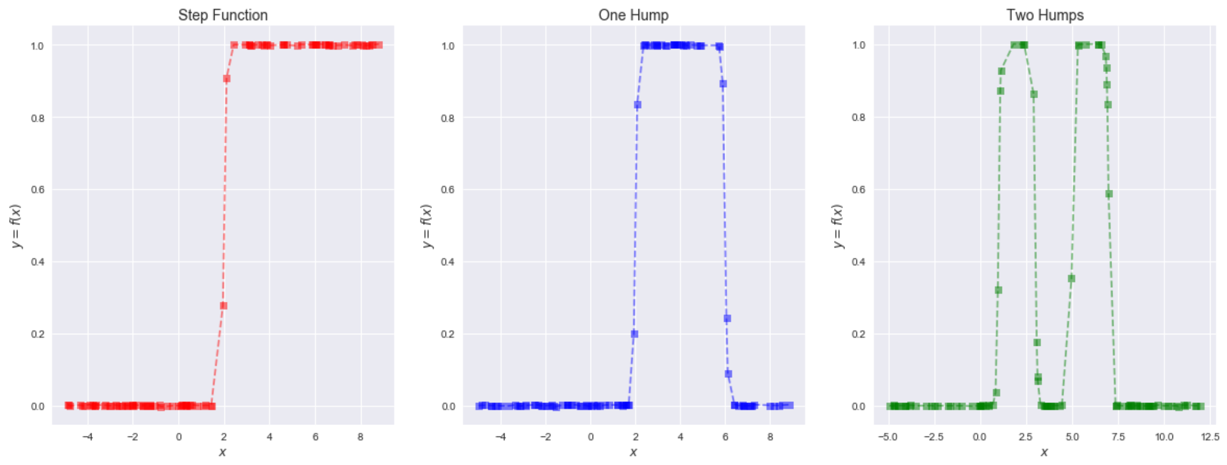
1.5 Choose the appropriate loss function and calculate and report the loss from all three cases. Derive the gradient of the output layer's weights for all three cases (step, one hump and two humps). Use the weights for the hidden layers you found in the previous question and perform gradient descent 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 [3]:
```python
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')

step_df = step_df.sort_values(by='x')
one_hump_df = one_hump_df.sort_values(by='x')
two_hump_df = two_hump_df.sort_values(by='x')
```

In [4]:
```python
# your code here
func = {'step_df': step_df, 'one_hump_df': one_hump_df, 'two_hump_df': two_hump_
fig_names = ['Step Function', 'One Hump', 'Two Humps']
col = ['red', 'blue','green']

fig, ax = plt.subplots(1, 3, figsize = (20, 7))
for i in range(len(list(func.keys()))):
    ax[i].plot(func[list(func.keys())[i]]['x'], func[list(func.keys())[i]]['y'],
               marker='s', ls='--', label='Step Function', alpha = 0.5)
    ax[i].set_xlabel(r'$x$', fontsize=12)
    ax[i].set_ylabel(r'$y=f(x)$', fontsize=12)
    ax[i].set_title(fig_names[i], fontsize=14)
```



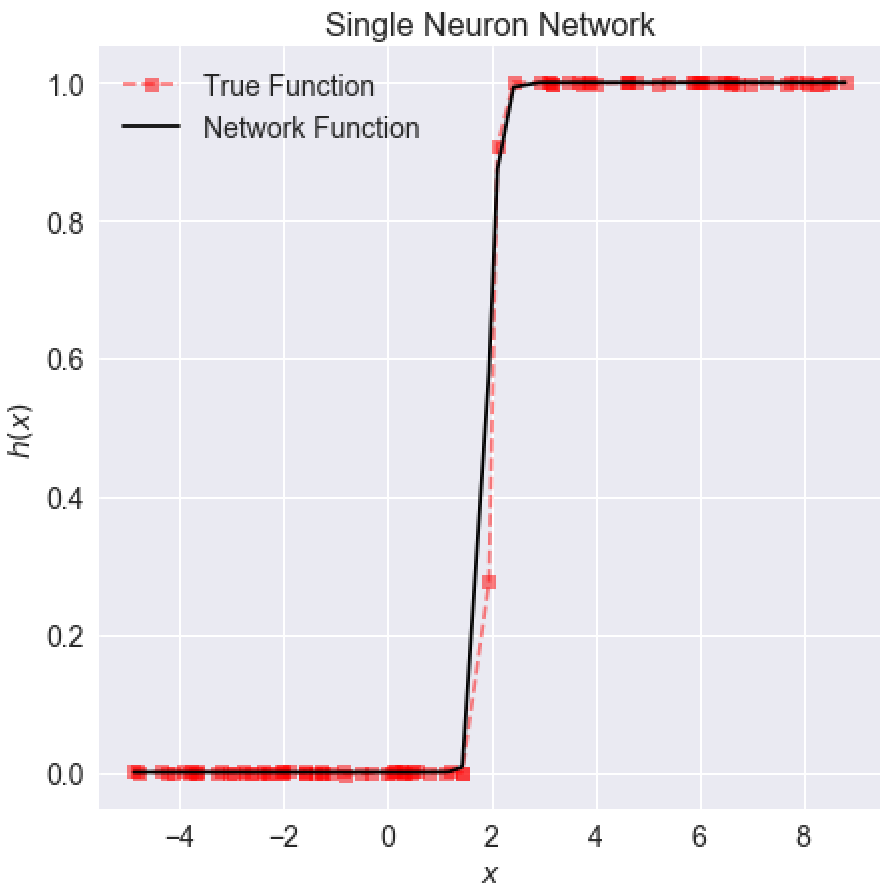
Answer: *The graph for the* `Step Function` *looks like a sigmoid, which means we can model this by directly using the output from the activation function from the hidden layer.* `One Hump` *and* `Two Humps` *look like a combination of sigmoids and thus, will require two and four nodes respectively.*

1.2

In [5]:
```
# your code here
def affine(w, x, b):
    z = w*x + b
    return(z)

def activate(z):
    h = 1.0 / (1.0 + np.exp(-z))
    return(h)

def yout(x, w, b, nodes, wout, bout):
    z = []
    for i in range(nodes):
        z.append(affine(w[i], x, b[i]))

    h = []
    for i in range(nodes):
        h.append(activate(z[i]))

    yout = bout
    for i in range(nodes):
        yout += wout[i]*h[i]

    return(yout, np.array(h))

def plot_network(x, true_y, predicted_y, title):
    fig, ax = plt.subplots(1, 1, figsize=(7,7))
    ax.plot(x, true_y, label=r'True Function', color='red', marker='s', ls='--',
    ax.plot(x, predicted_y, label=r'Network Function', color='k')
    ax.legend(loc='upper left', fontsize=14)
    ax.set_xlabel(r'$x$', fontsize=14)
    ax.set_ylabel(r'$h(x)$', fontsize=14)
    ax.set_title(title, fontsize=16)
    ax.tick_params(labelsize = 14)
```

In [6]:
```
x_step = step_df['x']
w_step = np.array([9.9])
b_step = np.array([-19])
wout_step = np.array([1])
bout_step = 0
nodes_step = 1

f_step = step_df['y']
yout_step, hout_step = yout(x=x_step, w=w_step, b=b_step, nodes=nodes_step,
                                       wout=wout_step, bout=bout_step)

# sanity check
print(hout_step.shape)

plot_network(x_step, f_step, yout_step, 'Single Neuron Network')
```

```
(1, 100)
```

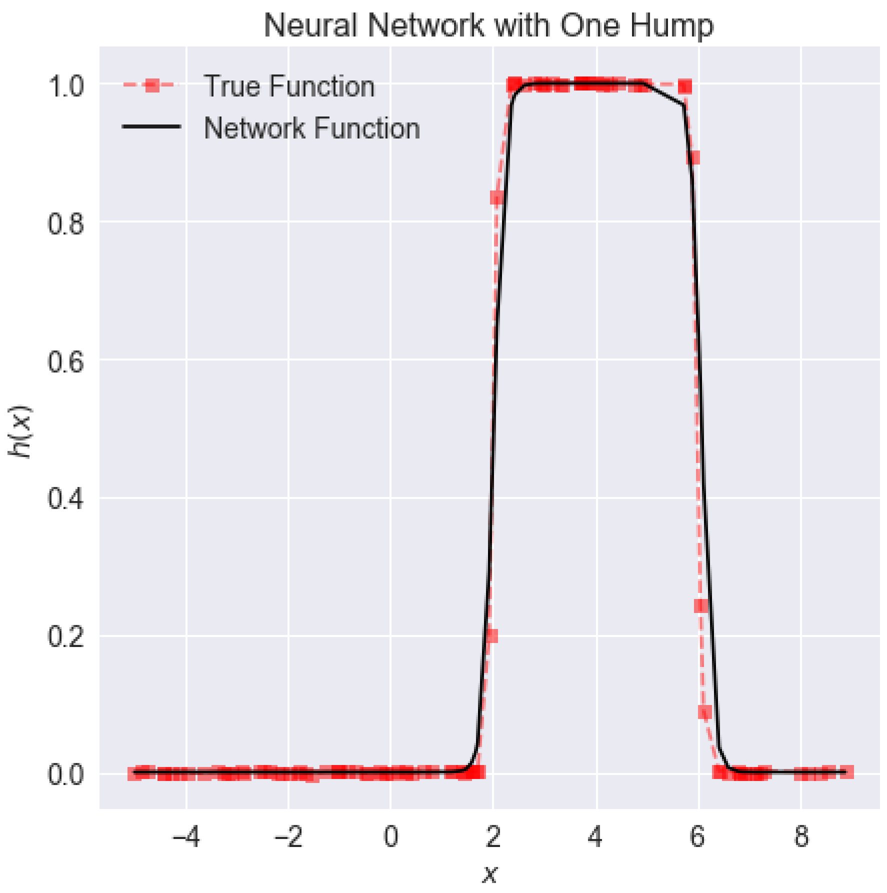


Answer: *Here we create functions* `affine` *,* `activate` *and* `yout` *that we can use for any number of nodes for given set of input x, hidden weights (w), hidden biases (b), output weights (wout), output biases (bout) and return the final output $Y hat$. Note that w, b and wout are arrays. After some trial and error we get above estimated function that closely matches out true repsonse.*

1.3

In [7]:
```
# your code here
x_one_hump = one_hump_df['x']
w_one_hump = np.array([9.9, -9.7])
b_one_hump = np.array([-20, 59])
wout_one_hump = np.array([1, 1])
bout_one_hump = -1
nodes_one_hump = 2

yout_one_hump, hout_one_hump = yout(x=x_one_hump, w=w_one_hump, b=b_one_hump, no
                                    wout=wout_one_hump, bout=bout_

# sanity check
print(hout_one_hump.shape)

f_one_hump = one_hump_df['y']

plot_network(x_one_hump, f_one_hump, yout_one_hump, 'Neural Network with One Hum
```

```
(2, 100)
```

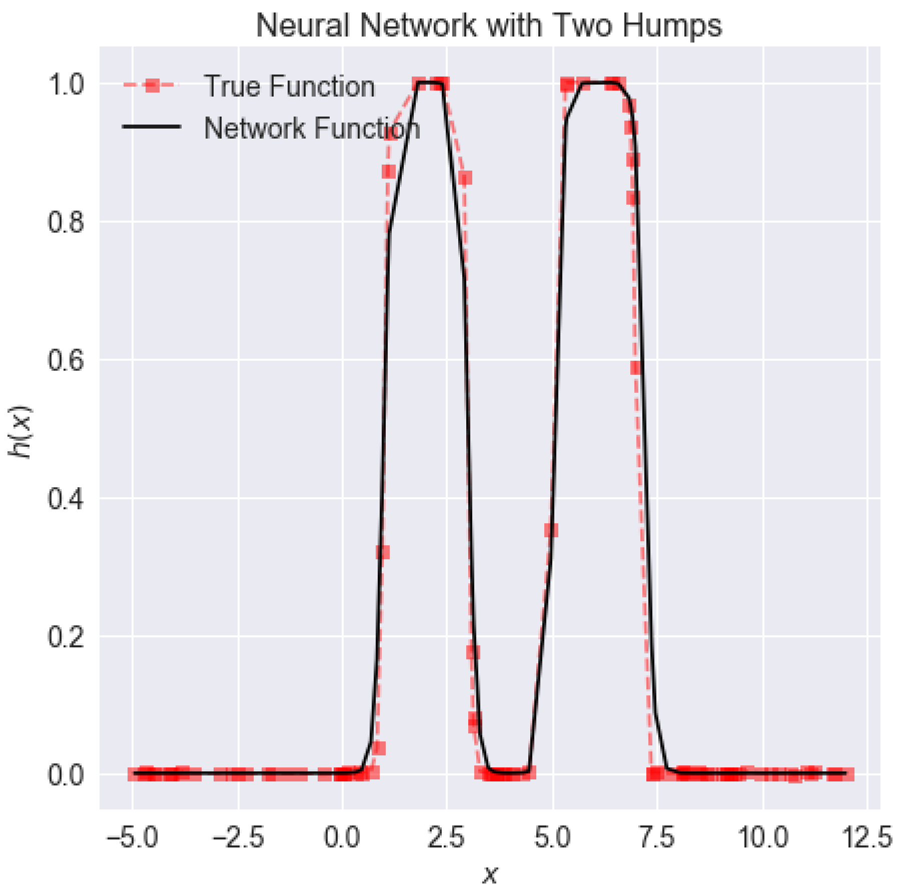


Answer: *We use the same process to get estimated function for* `One Hump` *data set. Observe that the hidden weights are opposite in sign and almost same in magnitude for both the nodes. Biases are also have opposite signs but different magnitudes, depending on where we want to "place" the second node's output.*

1.4

In [8]:
```
# your code here
x_two_hump = two_hump_df['x']
w_two_hump = np.array([10, -10, 9.9, -9.7])
b_two_hump = np.array([-10, 30, -50, 70])
wout_two_hump = np.array([1, 1, 1, 1])
bout_two_hump = -2
nodes_two_hump = 4

yout_two_hump, hout_two_hump = yout(x=x_two_hump, w=w_two_hump, b=b_two_hump, no
                                    wout=wout_two_hump, bout=bout_

# sanity check
print(hout_two_hump.shape)

f_two_hump = two_hump_df['y']

plot_network(x_two_hump, f_two_hump, yout_two_hump, 'Neural Network with Two Hum
```

```
(4, 100)
```



Answer: *Here we get four nodes with similar weight magnitudes and opposite signs for two consecutive nodes.*

1.5

In [9]:
```
# your code here
def CrossEntropy(yHat, y):
    eps = np.finfo(float).eps
    p = np.clip(yHat, eps, 1-eps)
    loss = -(y*np.log(p) + (1-y)*np.log(1-p))
    return(loss)
```

In [10]:
```
# Step function
error_step = CrossEntropy(yout_step, f_step)
display(sum(error_step))

# One Hump
error_one_hump = CrossEntropy(yout_one_hump, f_one_hump)
display(sum(error_one_hump))

# Two Humps
error_two_hump = CrossEntropy(yout_two_hump, f_two_hump)
display(sum(error_two_hump))
```

```
1.010605439499416

3.0761116377755724

7.16530007770289
```

Answer: *For calculating gradients of the output weights, we have to take the derivative of the loss function w.r.t. the output weights for each node. Since there is single output bias, bias gradient will always be a scalar irrespective of the number of nodes.*

Loss Function (Cross Entropy):

$$L = -\Big[y_i * logp + (1 - y_i) * log(1 - p)\Big]$$

where,

$$p = W_{out} . H_{out} + b_{out}$$

Derivative of the loss w.r.t. output layer weights:

$$\frac{d_L}{d_{wout}} = -h_2 \Big[\frac{y_i}{p} - \frac{(1 - y_i)}{(1 - p)}\Big]$$

Derivative of the loss w.r.t. output layer bias:

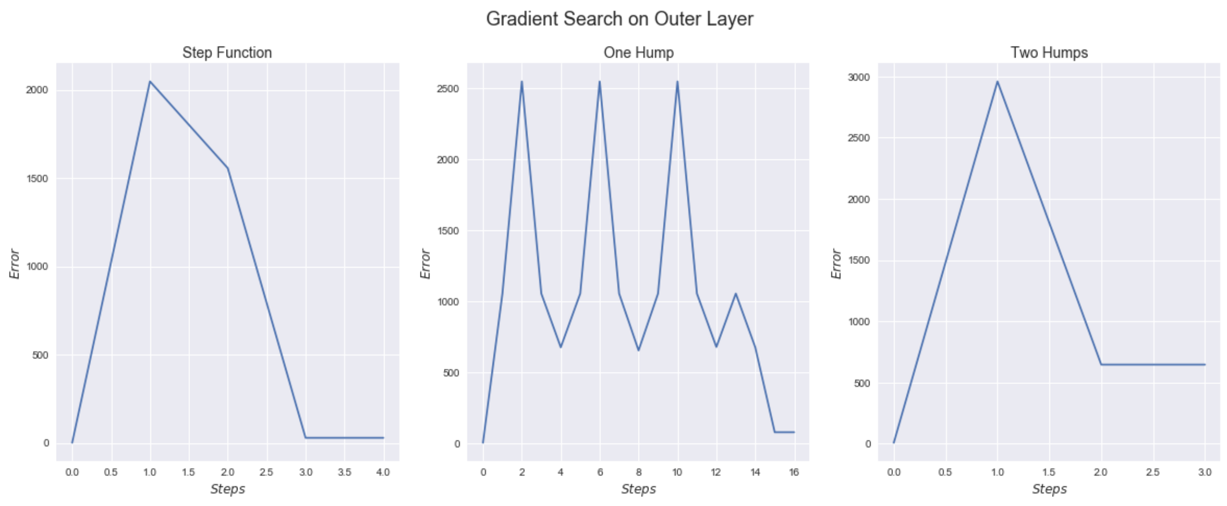
$$\frac{d_L}{d_{bout}} = -\Big[\frac{y_i}{p} - \frac{1 - y_i}{p}\Big]$$

In [11]:
```
# functions for gradient descent
def der_weight(x, y, w, b, nodes, wout, bout):
    y_out, h = yout(x, w, b, nodes, wout, bout)
    eps = np.finfo(float).eps
    p = np.clip(y_out, eps, 1-eps)
    dL_wout = np.dot(h, (-(y/p)+(1-y)/(1-p)))
    return(dL_wout)

def der_bias(yout, y):
    eps = np.finfo(float).eps
    p = np.clip(yout, eps, 1-eps)
    dL_bout = np.sum(-(y/p)+(1-y)/(1-p))
    return(dL_bout)

def gradient_descent(x, y, w, b, nodes, wout, bout, thresh=0.00001, l=0.001):
    y_out, hout = yout(x=x, w=w, b=b, nodes=nodes, wout=wout, bout=bout)
    errors = [np.sum(CrossEntropy(y_out, y))]
    wout_vals = [wout]
    bout_vals = [bout]

    for i in range(1, 1001):
        y_out, hout = yout(x, w, b, nodes, wout, bout)
        gradient_weight = der_weight(x, y, w, b, nodes, wout, bout)
        gradient_bias = der_bias(yout=y_out, y=y)

        wout_new = wout - l*gradient_weight
        bout_new = bout - l*gradient_bias

        yout_new, hout_new = yout(x, w, b, nodes, wout_new, bout_new)

        error = np.sum(CrossEntropy(yout_new, y))
        errors.append(error)
        wout_vals.append(wout_new)
        bout_vals.append(bout_new)

        if abs(error-errors[i-1])<thresh:
            i=i
            break
        else:
            wout = wout_new
            bout = bout_new
            i=i
    return(errors, wout_vals[-1], bout_vals[-1], i)
```

In [25]:
```
errors_grad_step, w_grad_step, b_grad_step, i_step = gradient_descent(x=x_step, y
                                                                     nodes=node
                                                                     bout=bout_
errors_grad_one, w_grad_one, b_grad_one, i_one = gradient_descent(x=x_one_hump, y
                                                                 b=b_one_hump, 
                                                                 wout=wout_one_
errors_grad_two, w_grad_two, b_grad_two, i_two = gradient_descent(x=x_two_hump, y
                                                                 b=b_two_hump, 
                                                                 wout=wout_two_


print('Wout for Step: %s' %w_grad_step)
print('Wout for One Hump: %s' %w_grad_one)
print('Wout for Two Hump: %s' %w_grad_two)

print(b_grad_step)
print(b_grad_one)
print(b_grad_two)

fig, ax = plt.subplots(1, 3, figsize=(20,7))
ax[0].plot(range(i_step+1), errors_grad_step)
ax[0].set_xlabel(r'$Steps$', fontsize=12)
ax[0].set_ylabel(r'$Error$', fontsize=12)
ax[0].set_title('Step Function', fontsize=14)


ax[1].plot(range(i_one+1), errors_grad_one)
ax[1].set_xlabel(r'$Steps$', fontsize=12)
ax[1].set_ylabel(r'$Error$', fontsize=12)
ax[1].set_title('One Hump', fontsize=14)

ax[2].plot(range(i_two+1), errors_grad_two)
ax[2].set_xlabel(r'$Steps$', fontsize=12)
ax[2].set_ylabel(r'$Error$', fontsize=12)
ax[2].set_title('Two Humps', fontsize=14)

fig.suptitle('Gradient Search on Outer Layer', y=0.98, fontsize = 18)
```

```
Wout for Step: [1.88852171e+14]
Wout for One Hump: [3.30994060e+14 1.57031116e+14]
Wout for Two Hump: [-1.46938059e+14 -1.20798938e+14 -1.07092700e+14 -1.43068467
e+14]
-65066801491440.9
-365158304750451.0
-288935616425653.1
```

Out[25]:
```
Text(0.5,0.98,'Gradient Search on Outer Layer')
```



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

In [13]:
```
# your code here
pima_df = pd.read_csv('data\pima-indians-diabetes.csv')
display(pima_df.describe())
display(pima_df.head(5))
display(pima_df.dtypes)
pima_df.isnull().sum()
```

	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	

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	A
0	6	148	72	35	0	33.6	0.627	
1	1	85	66	29	0	26.6	0.351	
2	8	183	64	0	0	23.3	0.672	
3	1	89	66	23	94	28.1	0.167	
4	0	137	40	35	168	43.1	2.288	

```
Pregnancies                   int64
Glucose                       int64
BloodPressure                 int64
SkinThickness                 int64
Insulin                       int64
BMI                         float64
DiabetesPedigreeFunction    float64
Age                           int64
Outcome                      object
dtype: object
```

Out[13]:
```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

In [14]:
```
# Cleaning the "Outcome" values
display(pima_df.Outcome.unique())
pima_df.Outcome = pima_df.Outcome.replace(to_replace={'1':1, '0':0, '0\\':0, '1\
pima_df.Outcome = pima_df.Outcome.astype('category')
pima_df.Outcome.unique()
```

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

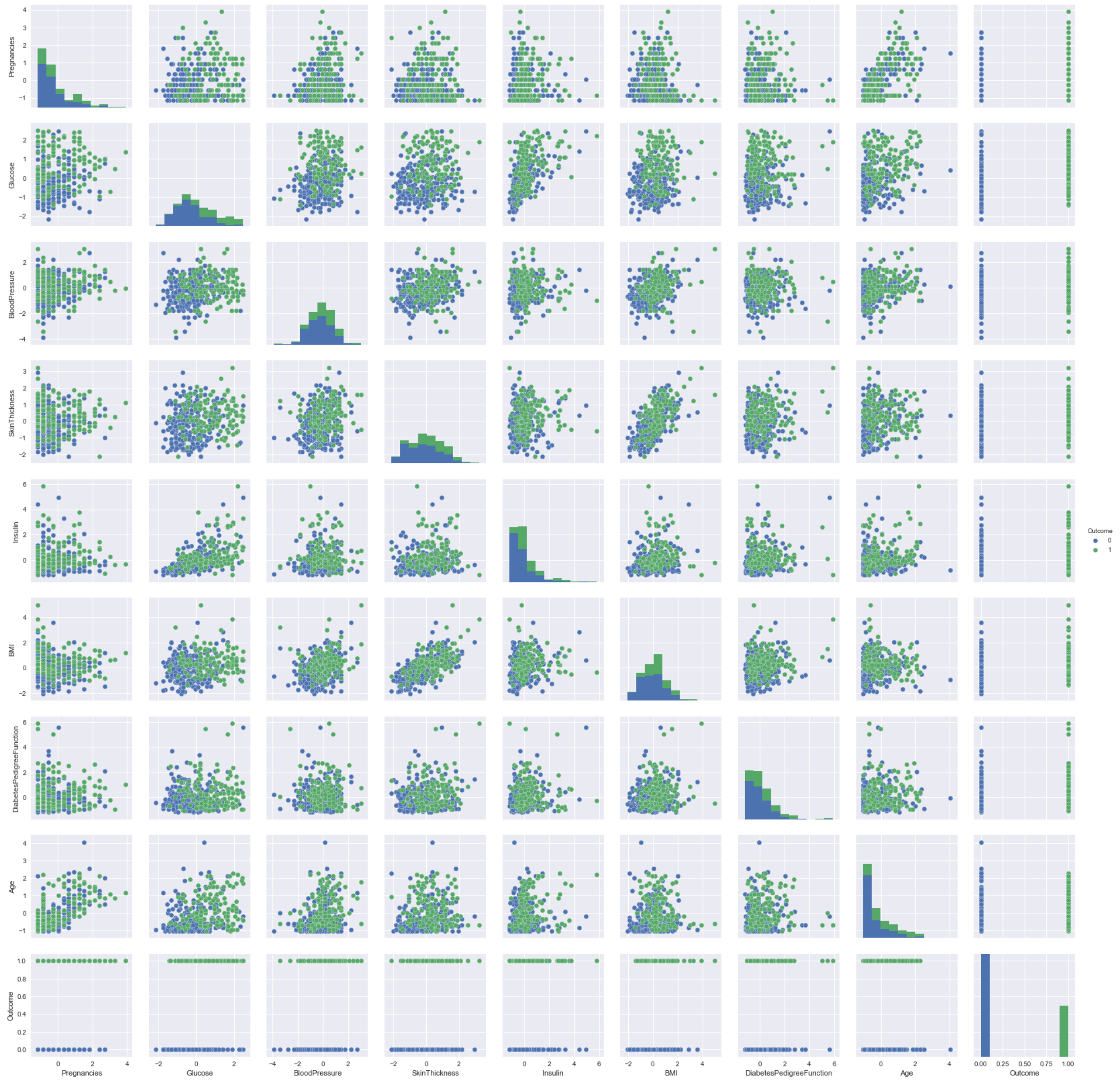
Out[14]:
```
[1, 0]
Categories (2, int64): [1, 0]
```

In [15]:
```
# functions for future preprocessing
def scale(fit_df, trans_df, na_col):
    df_scaled = trans_df.copy()
    for i in fit_df.columns:
        if i in na_col:
            fit_df_notna = fit_df[[i]].loc[fit_df[i]!=0]
            scaler = StandardScaler().fit(fit_df_notna)
            df_scaled.loc[df_scaled[i]!=0, i] = scaler.transform(df_scaled[[i]][
            df_scaled.loc[df_scaled[i]==0, i] = [np.nan]*len(df_scaled.loc[df_sca
        else:
            scaler = StandardScaler().fit(fit_df[[i]])
            df_scaled[i] = scaler.transform(df_scaled[[i]])
    return(df_scaled)

def Xy(df, response: str='Outcome'):
    practice_df = df.copy()
    y = practice_df[response]
    X = practice_df.drop(columns=response)
    return(y, X)

def impute_na(df_impute, na_col, method):
    df = df_impute.copy()
    for i in na_col:
        df.loc[df[i]==0, i] = [np.nan]*len(df.loc[df[i]==0, i])

    if method == 'none':
        df=df_impute
    elif method == 'mean':
        for i in list(df[na_col].columns):
            df.loc[df[i].isnull(), i] = df[i].dropna().mean()
    elif method in ['knn', 'lm']:
        predictors = list(set(na_col)^set(df_impute.columns))
        for i in list(df[na_col].columns):
            X_pred = df.loc[df[i].isnull(), predictors]
            y = df.loc[~df[i].isnull(), i]
            X = df.loc[~df[i].isnull(), predictors]
            X_scaled = scale(X, X, na_col = [])
            if method == 'knn':
                knn = KNeighborsRegressor(15).fit(X_scaled, y)
                df.loc[df[i].isnull(), i] = knn.predict(X_pred)
            elif method == 'lm':
                lm = LinearRegression().fit(X_scaled, y)
                df.loc[df[i].isnull(), i] = lm.predict(X_pred)
            predictors.append(i)
    return(df)

def get_design_mat(df, na_col, impute_method='none', response: list=['Outcome'])
    # impute values using 'impute_method'
    df_impute = impute_na(df, na_col=na_col, method=impute_method)

    train_df, test_df = train_test_split(df_impute, test_size=0.25, random_state

    # Get X,y
    y_train, X_train = Xy(train_df)
    y_test, X_test = Xy(test_df)
    # Scale
    X_train_scaled = scale(X_train, X_train, na_col)
    X_test_scaled = scale(X_train, X_test, na_col)

    return(X_train_scaled, y_train, X_test_scaled, y_test)
```

In [16]:
```
# Data Exploration
na_col = ['Glucose', 'BMI', 'BloodPressure', 'SkinThickness', 'Insulin']
pred_impute = []
response = ['Outcome']
pima_scaled=pima_df.copy()
scale_cols = list(pima_scaled.drop(columns=response).columns)
pima_scaled[scale_cols] = scale(pima_df[scale_cols], pima_df[scale_cols], na_col

# number of na's in each column
display(pima_scaled.isnull().sum())

# pairplot
pima_scaled = pima_scaled.dropna()
sns.pairplot(pima_scaled, hue='Outcome')
```

```
Pregnancies                   0
Glucose                       5
BloodPressure                35
SkinThickness               226
Insulin                     371
BMI                          11
DiabetesPedigreeFunction      0
Age                           0
Outcome                       0
dtype: int64
```

Out[16]:
```
<seaborn.axisgrid.PairGrid at 0x1b82f5ab38>
```



Answer:

1. We notice that the predictors `Glucose` , `BMI` , `BloodPressure` , `SkinThickness` and `Insulin` have a minimum value of 0, but based on some research online we find that they cannot never be 0. Hence we identify this set as the columns that contain disguised missing values.
2. Next, we notice that our response `Outcome` is an object in the dataset instead of being an integer or binary. So we do a `groupby()` on it to investigate further and find that it has bad data. We replace values that contain '0' as 0 and '1' as 1.
3. Then we write functions that go into `get_design_mats()` , where we will scale, get X and Y matrices, identify and impute missing values, and create train and test sets.
4. Notice that we get maximum number of missing values in the columns `SkinThickness` (226 NA's) and `Insulin` (371 NA's). Also notice in the `pairplot()` above that `SkinThickness` shows clear correlation to `BMI` and `Insulin` shows correlation `Glucose` . The observations in this point are important when we do model-based imputation in 2.4. We will explain our function `impute_na` there.

2.2

In [17]:
```
# get scaled matrices
X_train_scaled, y_train, X_test_scaled, y_test = get_design_mat(pima_df, na_col=
```

In [18]:
```
# drop na
X_train_dropna = X_train_scaled.dropna()
y_train_dropna = y_train[X_train_dropna.index]
X_test_dropna = X_test_scaled.dropna()
y_test_dropna = y_test[X_test_dropna.index]
logcv_sc = LogisticRegressionCV(cv=5, penalty='l2').fit(X_train_dropna, y_train_
print('Train accuracy score when NAs are dropped: %s'%accuracy_score(y_train_dro
print('Test accuracy score when NAs are dropped: %s'%accuracy_score(y_test_dropn
coef = {'Drop NAs': logcv_sc.coef_[0]}
```

```
Train accuracy score when NAs are dropped: 0.8
Test accuracy score when NAs are dropped: 0.7326732673267327
```

2.3

In [19]:
```
# your code here
X_train_mean, y_train, X_test_mean, y_test = get_design_mat(pima_df, na_col=na_c
logcv_avg = LogisticRegressionCV(cv=5, penalty='l2').fit(X_train_mean, y_train)
print('Train accuracy score when NAs are imputed with mean: %s'%accuracy_score(y_
print('Test accuracy score when NAs are imputed with mean: %s'%accuracy_score(y_
coef['Impute NAs - Mean'] = logcv_avg.coef_[0]
```

```
Train accuracy score when NAs are imputed with mean: 0.7713787085514834
Test accuracy score when NAs are imputed with mean: 0.7801047120418848
```

2.4

In [20]:
```
# your code here
# Missing values imputed using KNeighborsRegressor()
X_train_knn, y_train, X_test_knn, y_test = get_design_mat(pima_df, na_col=na_col
logcv_knn = LogisticRegressionCV(cv=5, penalty='l2').fit(X_train_knn, y_train)
print('Train accuracy score when NAs are imputed with KNeighborsRegressor(): %s'
print('Test accuracy score when NAs are imputed with KNeighborsRegressor(): %s'%
coef['Impute NAs - kNN'] = logcv_knn.coef_[0]
```

```
Train accuracy score when NAs are imputed with KNeighborsRegressor(): 0.7661431
064572426
Test accuracy score when NAs are imputed with KNeighborsRegressor(): 0.77486910
9947644
```

In [21]:
```
# Missing values imputes using LinearRegression()
X_train_lm, y_train, X_test_lm, y_test = get_design_mat(pima_df, na_col=na_col, 
logcv_lm = LogisticRegressionCV(cv=5, penalty='l2').fit(X_train_lm, y_train)
print('Train accuracy score when NAs are imputed with LinearRegression(): %s'%ac
print('Test accuracy score when NAs are imputed with LinearRegression(): %s'%acc
coef['Impute NAs - Linear Model'] = logcv_lm.coef_[0]
```

```
Train accuracy score when NAs are imputed with LinearRegression(): 0.7643979057
591623
Test accuracy score when NAs are imputed with LinearRegression(): 0.79057591623
03665
```

Answer:

- `impute_na()` : The function imputes NA's based on the method passed as an argument. Specifically for model-based methods (knn and lm) it starts with first variable in `na_cols` (list of variables that contain NA's, passed as an argument) and uses all the other columns that are not in this list to predict NA's. We keep adding each element in `na_cols` to our set of predictors as we go on imputing NA's.

2.5

Answer: *We have accuracy, corresponding to the method of imputation, in the descending order as: `lm` , `mean` , `knn` and drop NA's. `lm` does the best job as we have the two variables `SkinThickness` and `Insulin` that have maximum NA's to be linearly correlated with `BMI` and `Glucose` respectively - so we predict maximum missing values with highest accuracy with this method. Its interesting to see that `mean` does a slightly better job than `knn` - this is probably because of multi-collinearity. When we impute all the missing values for `SkinThickness` and `Insulin` with mean, our model thinks they are not important and other predictors (`BMI` and `Glucose` , as these are correlated to them) explain the variance instead. Dropping all observations with NA's is the most ineffective way as we lose useful information that could improve our results.*

In order to include as much information as possible to impute missing values, we make our list `na_cols` (that is passed to the function `impute_na()`) in the ascending order of the number of NA's so the variables with least NA's are imputed first and are used as predictors for imputing those with most NA's - so we predict the missing values of the variables with maximum number of NA's as accurately as possible with the available information.

2.6

In [22]:
```
pd.DataFrame(coef, index=X_train_scaled.columns)
```

Out[22]:

	Drop NAs	Impute NAs - Mean	Impute NAs - kNN	Impute NAs - Linear Model
Pregnancies	0.229560	0.412841	0.312723	0.378936
Glucose	0.734882	1.101539	0.828271	1.082110
BloodPressure	0.047397	-0.130239	-0.039781	0.105592
SkinThickness	0.135909	-0.016152	0.114843	-0.013337
Insulin	0.046651	-0.092583	0.048541	0.019272
BMI	0.314231	0.674449	0.470917	0.304163
DiabetesPedigreeFunction	0.221479	0.262457	0.222390	0.310531
Age	0.343882	0.150346	0.142781	0.118701

Answer: *Note that the variables that show highest change in their coefficients (* `Pregnancies` *,* `Age` *,* `BMI` *,* `Glucose` *) are the variables that have no or very few missing values. Which means that - A. we do lose out on valuable information when we drop all NA's and B. That the variables with highest missing values (* `SkinThickness` *and* `Insulin` *) are being regularized and replaced by variables correlated to them. Note that linear model imputation gives us coefficients closer to the ones with dropped NA's than any other model. Let's try to increase our regularization strength and see what we get.*

In [23]:
```
logcv_sc = LogisticRegressionCV(Cs=5, cv=5, penalty='l2').fit(X_train_dropna, y_
print('Train accuracy score when NAs are dropped: %s'%accuracy_score(y_train_drop
print('Test accuracy score when NAs are dropped: %s'%accuracy_score(y_test_dropna
coef5 = {'Drop NAs': logcv_sc.coef_[0]}

logcv_avg = LogisticRegressionCV(Cs=5, cv=5, penalty='l2').fit(X_train_mean, y_tr
print('Train accuracy score when NAs are imputed with mean: %s'%accuracy_score(y_
print('Test accuracy score when NAs are imputed with mean: %s'%accuracy_score(y_
coef5['Impute NAs - Mean'] = logcv_avg.coef_[0]

logcv_knn = LogisticRegressionCV(Cs=5, cv=5, penalty='l2').fit(X_train_knn, y_tra
print('Train accuracy score when NAs are imputed with KNeighborsRegressor(): %s'%
print('Test accuracy score when NAs are imputed with KNeighborsRegressor(): %s'%a
coef5['Impute NAs - kNN'] = logcv_knn.coef_[0]

logcv_lm = LogisticRegressionCV(Cs=5, cv=5, penalty='l2').fit(X_train_lm, y_trai
print('Train accuracy score when NAs are imputed with LinearRegression(): %s'%ac
print('Test accuracy score when NAs are imputed with LinearRegression(): %s'%acc
coef5['Impute NAs - Linear Model'] = logcv_lm.coef_[0]
```

```
Train accuracy score when NAs are dropped: 0.803448275862069
Test accuracy score when NAs are dropped: 0.7524752475247525
Train accuracy score when NAs are imputed with mean: 0.7713787085514834
Test accuracy score when NAs are imputed with mean: 0.7853403141361257
Train accuracy score when NAs are imputed with KNeighborsRegressor(): 0.7713787
085514834
Test accuracy score when NAs are imputed with KNeighborsRegressor(): 0.78534031
41361257
Train accuracy score when NAs are imputed with LinearRegression(): 0.7643979057
591623
Test accuracy score when NAs are imputed with LinearRegression(): 0.79057591623
03665
```

In [24]:
```
pd.DataFrame(coef5, index=X_train_scaled.columns)
```

Out[24]:

	Drop NAs	Impute NAs - Mean	Impute NAs - kNN	Impute NAs - Linear Model
Pregnancies	0.244598	0.428384	0.422887	0.390673
Glucose	1.254742	1.142494	1.094012	1.114568
BloodPressure	-0.049552	-0.143579	-0.113211	0.108129
SkinThickness	0.045572	-0.029456	0.068624	-0.013340
Insulin	-0.241585	-0.109817	-0.025156	0.016736
BMI	0.611585	0.703629	0.632550	0.308367
DiabetesPedigreeFunction	0.328791	0.269282	0.266481	0.317155
Age	0.569583	0.148776	0.135216	0.112601

Answer: *We get the same accuracy for* `Impute NAs - Linear Model` *as before, while the* *accuracy for `Impute NAs - kNN` is slightly increased and is now the same as `Impute NAs - Mean`. Also note how `SkinThickness` and `Insulin` are almost completely regularized in `Impute NAs - Linear Model`.*

In [ ]: