Important Note for question1 !

- Please **do not** change the default variable names in this problem, as we will use them in different parts.
- The default variables are initially set to "None".
- You only need to modify code in the "TODO" part. We added a lot of "assertions" to check your code. **Do not** modify them.

In [2]:
```
# Load packages
import numpy as np
import pandas as pd
import math
import time
from sklearn.naive_bayes import GaussianNB
```

P1. Load data and plot

TODO

- Load train and test data, and split them into inputs(trainX, testX) and labels(trainY, testY)

In [3]:
```
# Use pandas to load q1_train.csv and q1_test.csv
# Each data point has 200 features(X), followed by 1 label(Y)

#### TODO ####
train = pd.read_csv("q1_train.csv")
trainnp = train.to_numpy()
trainX = trainnp[:,1:201]
trainY = trainnp[:,201]
test = pd.read_csv("q1_test.csv")
testnp = test.to_numpy()
testX = testnp[:,1:201]
testY = testnp[:,201]
##############

assert(len(trainX.shape) == 2)
assert(len(trainY.shape) == 1)
assert(trainX.shape[1] == 200)
```

P2. Write your Gaussian NB solver

TODO

- Finish the myNBSolver() function.
  - Compute P(y == 0) and P(y == 1), saved in "py0" and "py1"

- Compute mean/variance of trainX for both y = 0 and y = 1, saved in "mean0", "var0", "mean1" and "var1"
    - Each of them should have shape (M), where M is number of features.
- Compute P(xi | y == 0) and P(xi | y == 1), compare and save **binary** prediction in "train_pred" and "test_pred"
- Compute train accuracy and test accuracy, saved in "train_acc" and "test_acc".
- Return train accuracy and test accuracy.

In [4]:
```
def myNBSolver(trainX, trainY, testX, testY):
    N_train = trainX.shape[0]

    N_train = trainX.shape[0]
    N_test = testX.shape[0]
    M = trainX.shape[1]

    #### TODO ####
    # Compute P(y == 0) and P(y == 1)
    py0 = (trainY.tolist().count(0))/N_train
    py1 = (trainY.tolist().count(1))/N_train

    ##############
    print("Total probablity is %.2f. Should be equal to 1." %(py0 + py1))

    #Find 0s
    zerolocationstuple = np.where(trainY == 0)
    zerolocations, = zerolocationstuple
    #Find 1s
    onelocationtuple = np.where(trainY == 1)
    onelocations, = onelocationtuple

    trainX1 = []
    trainX0 = []

    for value in onelocations:
        trainX1.append(trainX[value, :])

    trainX1 = np.array(trainX1)

    for value in zerolocations:
        trainX0.append(trainX[value,:])

    trainX0 = np.array(trainX0)

    meanlist0 = []
    meanlist1 = []
    varlist0 = []
    varlist1 = []

    M = trainX.shape[1]

    for feature in range(M):
        Train1 = trainX1[:,feature]
        mean1 = np.mean(Train1)
        var1 = np.var(Train1)
        meanlist1.append(mean1)
        varlist1.append(var1)

        Train0 = trainX0[:,feature]
        mean0 = np.mean(Train0)
        var0 = np.var(Train0)
        meanlist0.append(mean0)
        varlist0.append(var0)

    ## TODO ####
```

```python
# Compute mean/var for each label

mean0 = np.array(meanlist0)
mean1 = np.array(meanlist1)
var0 = np.array(varlist0)
var1 = np.array(varlist1)

##############
assert(mean0.shape[0] == M)
print("Mean and Var Calculated")

#### TODO ####
# Compute P(xi|y == 0) and P(xi|y == 1), compare and make prediction
# This part may spend 5 - 10 minutes or even more if you use for loop, so fee
# print something (like step number) to check the progress

ProbX0all = []
ProbX1all = []

print("Calculating Probablity for Training Sample")
for sample in range(N_train):
    ProbX0 = []
    ProbX1 = []
    print(sample)
    for feature in range(M):
        prob0 = (1/((2*math.pi*(var0[feature]**2))**0.5))*math.exp(-(((train
        ProbX0.append(prob0)
        ProbX0np = np.array(ProbX0)
        prob1 = (1/((2*math.pi*(var1[feature]**2))**0.5))*math.exp(-(((train
        ProbX1.append(prob1)
        ProbX1np = np.array(ProbX1)

    ProbX0all.append(np.prod(ProbX0np)*py0)
    ProbX1all.append(np.prod(ProbX1np)*py1)

ProbX0allnp = np.array(ProbX0all)
ProbX1allnp = np.array(ProbX1all)

correctlist = []
missedlist = []
elselist = []

#train_pred = np.concatenate((ProbX0allnp, ProbX1allnp), axis=0)
train_pred_list = []

print("Analyzing Training Sample")
for trainvalue in range(len(ProbX0all)):
    print(trainvalue)
    if ProbX0all[trainvalue] > ProbX1all[trainvalue]:
        if trainY[trainvalue] == 0:
            correctlist.append(1)
            train_pred_list.append(0)
        else:
            missedlist.append(1)
            train_pred_list.append(0)

    elif ProbX0all[trainvalue] < ProbX1all[trainvalue]:
```

```
            if trainY[trainvalue] == 1:
                correctlist.append(1)
                train_pred_list.append(1)
            else:
                missedlist.append(1)
                train_pred_list.append(1)
        else:
            elselist.append(1)
            print(tie)

    accuracy = (len(correctlist)/N_train)
    print("Train")
    print(accuracy)

    #####################################################################3
    ProbX0all_test = []
    ProbX1all_test = []

    test_pred_list = []
    print("Calculating Test Sample Probability")
    for sample_test in range(N_test):
        print(sample_test)
        ProbX0_test = []
        ProbX1_test = []
        for feature in range(M):
            prob0_test = (1/((2*math.pi*(var0[feature]**2))**0.5))*math.exp(-(((
            ProbX0_test.append(prob0_test)
            ProbX0np_test = np.array(ProbX0_test)
            prob1_test = (1/((2*math.pi*(var1[feature]**2))**0.5))*math.exp(-(((
            ProbX1_test.append(prob1_test)
            ProbX1np_test = np.array(ProbX1_test)

        ProbX0all_test.append(np.prod(ProbX0np_test)*py0)
        ProbX1all_test.append(np.prod(ProbX1np_test)*py1)

    ProbX0allnp_test = np.array(ProbX0all_test)
    ProbX1allnp_test = np.array(ProbX1all_test)

    correctlist_test = []
    missedlist_test = []
    elselist = []


    print("Analyzing Test Sample")
    for testvalue in range(len(ProbX0all_test)):
        if ProbX0all_test[testvalue] > ProbX1all_test[testvalue]:
            if testY[testvalue] == 0:
                correctlist_test.append(1)
                test_pred_list.append(0)
            else:
                missedlist_test.append(1)
                test_pred_list.append(0)
        elif ProbX0all_test[testvalue] < ProbX1all_test[testvalue]:
            if testY[testvalue] == 1:
                correctlist_test.append(1)
                test_pred_list.append(1)
            else:
```

```
                missedlist_test.append(1)
                test_pred_list.append(1)
        else:
            elselist.append(1)
            print(tie)




    accuracy_test = (len(correctlist_test)/N_test)
    print("Test")
    print(accuracy_test)

     #### TODO ####
    # Compute train accuracy and test accuracy

    accuracy = (len(correctlist)/N_train)
    print("Train")
    print(accuracy)

    accuracy_test = (len(correctlist_test)/N_test)
    print("Test")
    print(accuracy_test)

    train_acc = accuracy
    test_acc = accuracy_test

    return train_acc,test_acc
```

In [5]:

```
# driver to test your NB solver
train_acc, test_acc = myNBSolver(trainX, trainY, testX, testY)
print("Train accuracy is %.2f" %(train_acc * 100))
print("Test accuracy is %.2f" %(test_acc * 100))
```

```
65990
65991
65992
65993
65994
65995
65996
65997
65998
65999
Analyzing Test Sample
Test
0.919030303030303
Train
0.9206716417910448
Test
0.919030303030303
Train accuracy is 92.07
Test accuracy is 91.90
```

P3. Test your result using sklearn

TODO

- Finish the skNBSolver() function.
  - fit model, make prediction and return accuracy for train and test sets.

In [6]:
```
def skNBSolver(trainX, trainY, testX, testY):

    #### TODO ####
    # fit model
    # make prediction
    # compute accuracy
    GNB = GaussianNB()
    GNB.fit(trainX,trainY)
    sk_train_acc = GNB.score(trainX,trainY)
    sk_test_acc = GNB.score(testX,testY)

    ##############
    return sk_train_acc, sk_test_acc
```

In [7]:
```
# driver to test skNBSolver
sk_train_acc, sk_test_acc = skNBSolver(trainX, trainY, testX, testY)
print("Train accuracy is %.2f" %(sk_train_acc * 100))
print("Test accuracy is %.2f" %(sk_test_acc * 100))
```

```
Train accuracy is 92.22
Test accuracy is 92.05
```

Note for question2

- Please follow the template to complete q2
- You may create new cells to report your results and observations

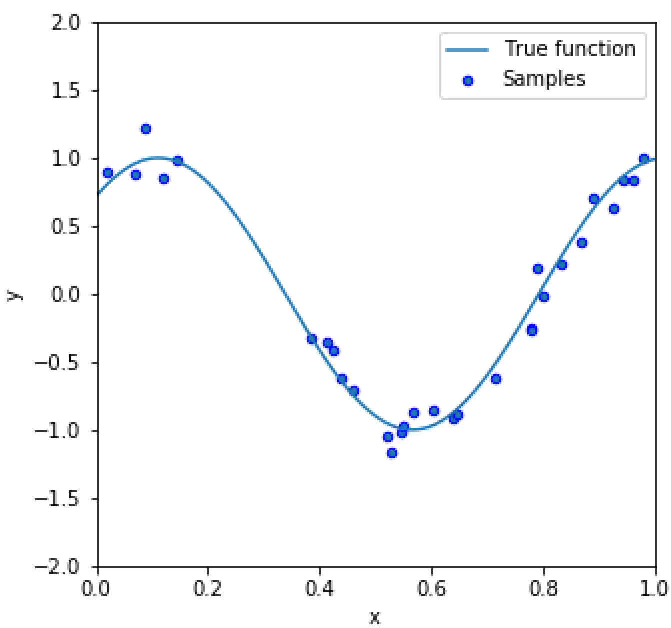
In [3]:
```
# Import modules
import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
```

P1. Create data and plot

TODO

- implement the true function $f(x)$ defined in the write-up
- use function name **model()**
- sample 30 random points with noise
- plot sampled points together with the model function

In [4]:
```python
# Define the function to generate data points
def model(X):
    return np.sin(2.2 * np.pi * X + 0.8)
# Initialize random seed
np.random.seed(0)
# Generate noisy data points: (x,y)
n_samples = 30
x = np.sort(np.random.rand(n_samples))
y = model(x) + np.random.randn(n_samples) * 0.1
# Plot true model and sampled data points
plt.figure(figsize=(5, 5))
X_test = np.linspace(0, 1, 100)
plt.plot(X_test, model(X_test), label="True function")
plt.scatter(x, y, edgecolor='b', s=20, label="Samples")
plt.xlabel("x")
plt.ylabel("y")
plt.xlim((0, 1))
plt.ylim((-2, 2))
# Visualize data points
plt.legend(loc="best")
plt.show()
```



P2. Fit a linear model

TODO

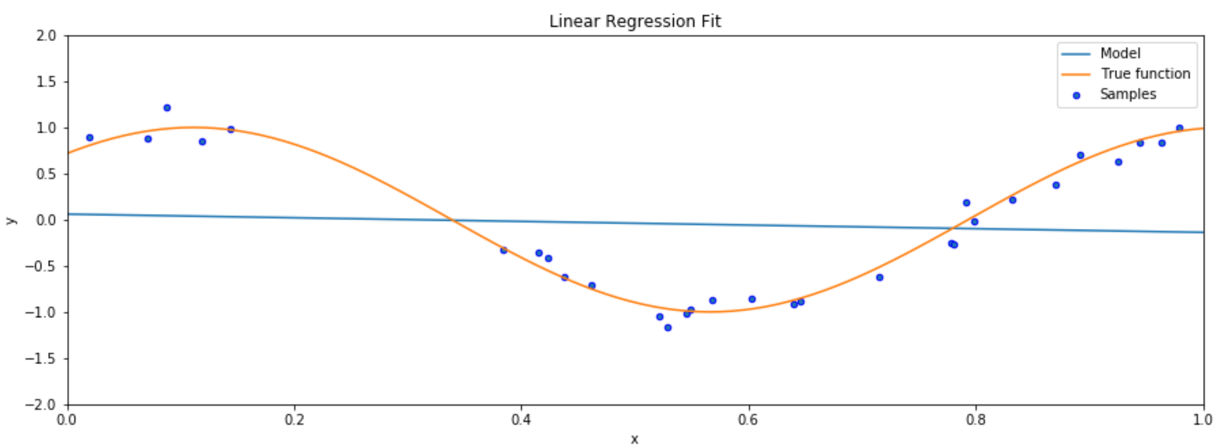
- use sklearn to fit model: $h(x) = w_0 + w_1 x$
- report $w = [w_0, w_1]$

- plot the fitted model $h(x)$ together with data points

In [5]:
```python
# Fit a linear model in the original space
x = x.reshape(-1,1)
y = y.reshape(-1,1)
reg = LinearRegression().fit(x, y)
w1 = reg.coef_[0]
w0 = reg.intercept_
w = [w0,w1]
print(w)
```

```
[array([0.06038094]), array([-0.19787027])]
```

In [6]:
```python
# Plot fitted linear model
plt.figure(figsize=(15, 5))
X_test = np.linspace(0, 1, 100)
plt.plot(X_test, reg.predict(X_test[:, np.newaxis]), label="Model")
plt.plot(X_test, model(X_test), label="True function")
plt.scatter(x, y, edgecolor='b', s=20, label="Samples")
plt.xlabel("x")
plt.ylabel("y")
plt.xlim((0, 1))
plt.ylim((-2, 2))
plt.legend(loc="best")
plt.title("Linear Regression Fit")
plt.show()
```



P3. Fit a polynomial curve

TODO

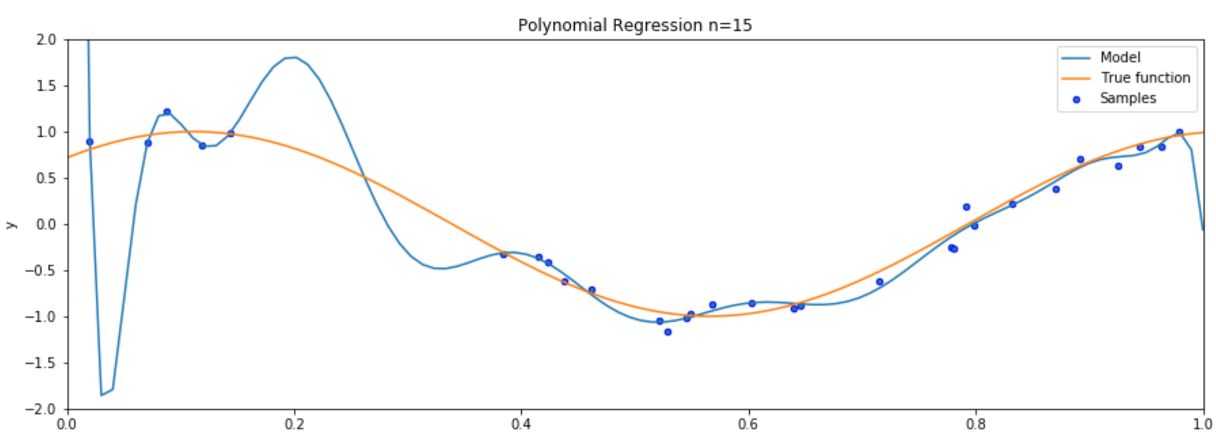
- augment the original feature to $[x, x^2, \cdots, x^{15}]$
- fit the polynomial curve: $h(x) = \sum_{i=0}^{15} w_i x^i$
- report $w = [w_0, w_1, \cdots, w_{15}]$
- plot the fitted model $h(x)$ together with data points

In [7]:
```
# Augment the original feature to a 15-vector
degree = 15
n15reg = PolynomialFeatures(degree)
augx = n15reg.fit_transform(x)
print(augx.shape)
# Fit linear model to the generated 15-vector features
augreg = LinearRegression().fit(augx,y)
w = augreg.coef_[0]
weights = np.insert(w,0,augreg.intercept_[0])
print(weights)
```

```
(30, 16)
[ 3.11666317e+01  0.00000000e+00 -2.97809480e+03  1.03892675e+05
 -1.87418803e+06  2.03715545e+07 -1.44872449e+08  7.09311984e+08
 -2.47064769e+09  6.24558698e+09 -1.15676113e+10  1.56894446e+10
 -1.54005585e+10  1.06456986e+10 -4.91376344e+09  1.35919341e+09
 -1.70380431e+08]
```

In [11]:
```
# Plot fitted curve and sampled data points
from sklearn.pipeline import make_pipeline
n15reg=make_pipeline(PolynomialFeatures(15),LinearRegression())
n15reg.fit(x,y)
plt.figure(figsize=(15, 5))
X_test = np.linspace(0, 1, 100)
yval = n15reg.predict(X_test[:, np.newaxis])
print(yval.shape)
plt.plot(X_test, n15reg.predict(X_test[:, np.newaxis]), label="Model")
plt.plot(X_test, model(X_test), label="True function")
plt.scatter(x, y, edgecolor='b', s=20, label="Samples")
plt.xlabel("x")
plt.ylabel("y")
plt.xlim((0, 1))
plt.ylim((-2, 2))
plt.legend(loc="best")
plt.title("Polynomial Regression n=15")
plt.show()
```

```
(100, 1)
```



P4. Lasso regularization

TODO

- use sklearn to fit a 15-degree polynomial model with L1 regularization
- report w
- plot the fitted model $h(x)$ together with data points

In [18]:

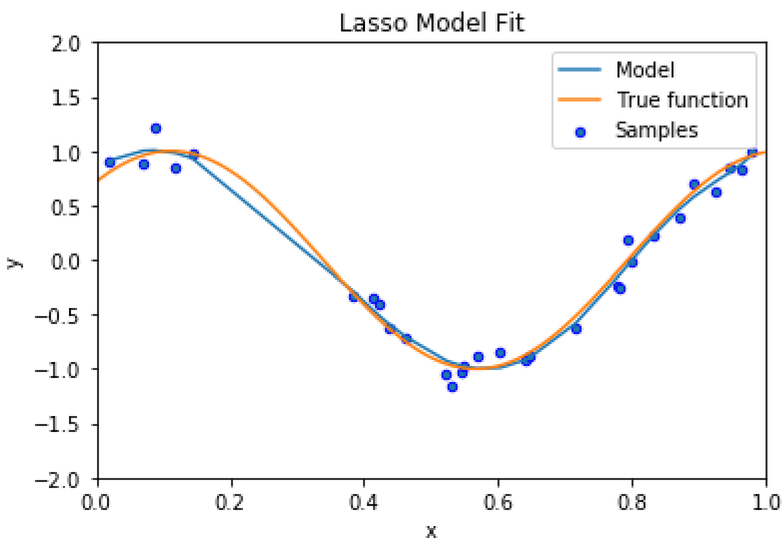
```
# Fit 15-degree polynomial with L1 regularization
# Start with lambda(alpha) = 0.01 and max_iter = 1e4
from sklearn import linear_model
LassoModel = linear_model.Lasso(alpha=0.000007, max_iter=1e4)
LassoModel.fit(augx,y)
w = LassoModel.coef_
weights = np.insert(w,0,LassoModel.intercept_[0])
print(weights)
```

```
[  0.84115451   0.           4.04940069 -26.18760545  15.00658696
  13.26518055   5.31423695   0.          -2.1566783   -4.65298778
  -4.52580357  -3.44643585  -1.92253837  -0.20761126  -0.
   1.16084506   4.60247859]
```

```
C:\Users\rdesa\Anaconda3\lib\site-packages\sklearn\linear_model\coordinate_desc
ent.py:475: ConvergenceWarning: Objective did not converge. You might want to i
ncrease the number of iterations. Duality gap: 0.09454608208037706, tolerance:
0.00175155385437781
  positive)
```

In [13]:

```
# Plot fitted curve and sampled data points
from sklearn.linear_model import Lasso
X_test = np.linspace(0, 1, 20)
plt.plot(x, LassoModel.predict(augx), label="Model")
X_test = np.linspace(0, 1, 100)
plt.plot(X_test, model(X_test), label="True function")
plt.scatter(x, y, edgecolor='b', s=20, label="Samples")
plt.xlabel("x")
plt.ylabel("y")
plt.xlim((0, 1))
plt.ylim((-2, 2))
plt.legend(loc="best")
plt.title("Lasso Model Fit")
plt.show()
```



In [ ]:

```
#Observation of values w and lambda
# The best lambda value was very small 0.000007
# The only 3 weights were eliminated to get the
#model to fit properly the weights on x1, x1^3,
#and x1^5 were set to zero
#Understanding of Lasso Regularization
#Lasso Regression removes weights causing
#overfitting by shrinking unnessary thetas by
#finding the vertex of the diamond
#However too many varibles can be eliminated
#if lambda is set to high becasue the lasso
#misses the vertex of the diamond
#When completed properly only the weights
#on the most important values should
#remain
#Observation when lambda is tweaked
#The larger lambda eliminated more weights
#The larger lambda would eliminate too many
#weights causing the model to underfit, only
#when the lambda was sufficiently small to
#eliminate only the weights that caused
#underfitting did the lasso regression fit
#the real model well
```

P5. Ridge regularization

TODO

- use sklearn to fit a 15-degree polynomial model with L2 regularization
- report w
- plot the fitted model $h(x)$ together with data points

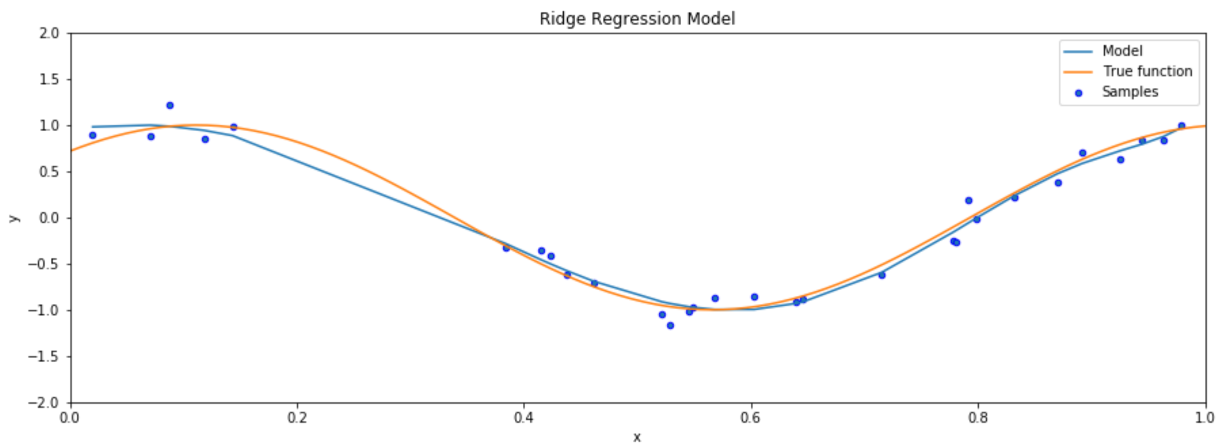
In [14]:

```
# Fit 15-degree polynomial with L2 regularization
# Start with lambda(alpha) = 0.01 and max_iter = 1e4
from sklearn.linear_model import Ridge
RidgeModel = Ridge(alpha= 0.0001, max_iter=1e5)
RidgeModel.fit(augx,y)
print(RidgeModel.coef_)
print(RidgeModel.intercept_)
```

```
[[  0.           1.90502172 -17.24276774   4.25002429  11.61238507
    9.20596081   3.83848301  -1.02446738  -4.18410939  -5.53450391
   -5.38543602  -4.13771324  -2.15164759   0.29118807   2.98672066
    5.79387602]]
[0.94933395]
```

In [15]:
```
# Plot fitted curve and sampled data points and compare to L1 regularization from
plt.figure(figsize=(15, 5))
X_test = np.linspace(0, 1, 100)
#LassoModel.predict(X_test[:, np.newaxis])
print(augx.shape)
print(x.shape)
plt.plot(x, RidgeModel.predict(augx), label="Model")
plt.plot(X_test, model(X_test), label="True function")
plt.scatter(x, y, edgecolor='b', s=20, label="Samples")
plt.xlabel("x")
plt.ylabel("y")
plt.xlim((0, 1))
plt.ylim((-2, 2))
plt.legend(loc="best")
plt.title("Ridge Regression Model")
plt.show()
```

```
(30, 16)
(30, 1)
```



In [ ]:

Note for question3

- Please follow the template to complete q3
- You may create new cells to report your results and observations

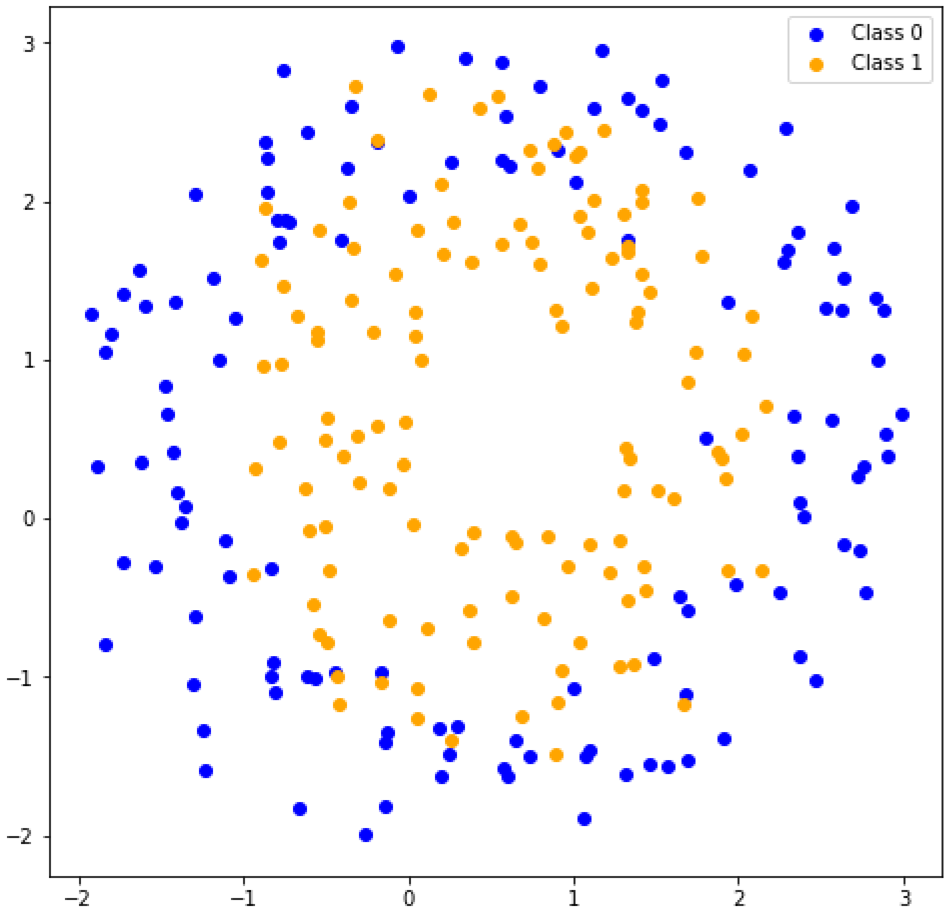
In [3]:
```
# Import libraries
import csv
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (10.0, 6.0)
from mpl_toolkits.mplot3d import Axes3D
import time
import math
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import PolynomialFeatures
import random
```

P1. Load data and plot

TODO

- load q3_data.csv
- plot the points of different labels with different color

In [4]:
```
# Load dataset
data = pd.read_csv("q3_data.csv")
datanp = data.to_numpy()
x = datanp[:,0:2]
label = datanp[:,2]

#plot
plt.figure(figsize = (8,8))
plt.scatter(x[:126,0], x[:126,1], c = 'blue', label="Class 0")
plt.scatter(x[127:,0], x[127:,1], c = 'orange', label ="Class 1")
plt.legend()
plt.show()
```



P2. Feature mapping

TODO

- implement function **map_feature()** to transform data from original space to the 28D space specified in the write-up

In [5]:
```
print(x.shape)
samples, Dimension = x.shape
assert(Dimension==2)
```

```
(251, 2)
```

In [6]:
```
# Transform points to 28D space
def map_feature(x, degree=6):
    x2D = x
    samples, Dimension = x2D.shape
    assert(Dimension==2)
    poly = PolynomialFeatures(degree)
    x28D = poly.fit_transform(x)
    return x28D
x28D = map_feature(x, degree=6)
print(x28D.shape)
```

```
(251, 28)
```

P3. Regularized Logistic Regression

TODO

- implement function **logistic_regpression_regularized()** as required in the write-up
- draw the decision boundary

Hints

- recycling code from HW2 is allowed
- you may use functions defined this section for part 4 below
- although optional for the report, plotting the convergence curve will be helpful

In [7]:
```
X = x28D
Y = label
print(X.shape)
print(Y.shape)
```

```
(251, 28)
(251,)
```

In [8]:
```
# Pass in the required arguments
# Implement the sigmoid function
def sigmoid(z):
    return 1/(1+np.exp(-z))
```

In [9]:
```
def Pred(w,X):
    z = np.array(w[0]+w[1]*np.array(X[:,0])
             +w[2]*np.array(X[:,1])+w[3]*np.array(X[:,2])
             +w[4]*np.array(X[:,3])+w[5]*np.array(X[:,4])
             +w[6]*np.array(X[:,5])+w[7]*np.array(X[:,6])
             +w[8]*np.array(X[:,7])+w[9]*np.array(X[:,8])
             +w[10]*np.array(X[:,9])+w[11]*np.array(X[:,10])
             +w[12]*np.array(X[:,11])+w[13]*np.array(X[:,12])
             +w[14]*np.array(X[:,13])+w[15]*np.array(X[:,14])
             +w[16]*np.array(X[:,15])+w[17]*np.array(X[:,16])
             +w[18]*np.array(X[:,17])+w[19]*np.array(X[:,18])
             +w[20]*np.array(X[:,19])+w[21]*np.array(X[:,20])
             +w[22]*np.array(X[:,21])+w[23]*np.array(X[:,22])
             +w[24]*np.array(X[:,23])+w[25]*np.array(X[:,24])
             +w[26]*np.array(X[:,25])+w[27]*np.array(X[:,26]))
    return sigmoid(z)
```

In [10]:
```
def calculate_gradients(w, X, Y, lamb):
        pred = Pred(w,X)
        m,feat = X.shape
        gradient = [0]*28
        gradient[0] = -1 * sum(Y*(1-pred) - (1-Y)*pred)
        gradient[1] = -1 * (sum(Y*(1-pred)*X[:,0] - (1-Y)*pred*X[:,0])+(lamb*w[1
        gradient[2] = -1 * (sum(Y*(1-pred)*X[:,1] - (1-Y)*pred*X[:,1])+(lamb*w[2
        gradient[3] = -1 * (sum(Y*(1-pred)*X[:,2] - (1-Y)*pred*X[:,2])+(lamb*w[3
        gradient[4] = -1 * (sum(Y*(1-pred)*X[:,3] - (1-Y)*pred*X[:,3])+(lamb*w[4
        gradient[5] = -1 * (sum(Y*(1-pred)*X[:,4] - (1-Y)*pred*X[:,4])+(lamb*w[5
        gradient[6] = -1 * (sum(Y*(1-pred)*X[:,5] - (1-Y)*pred*X[:,5])+(lamb*w[6
        gradient[7] = -1 * (sum(Y*(1-pred)*X[:,6] - (1-Y)*pred*X[:,6])+(lamb*w[7
        gradient[8] = -1 * (sum(Y*(1-pred)*X[:,7] - (1-Y)*pred*X[:,7])+(lamb*w[8
        gradient[9] = -1 * (sum(Y*(1-pred)*X[:,8] - (1-Y)*pred*X[:,8])+(lamb*w[9
        gradient[10] = -1 * (sum(Y*(1-pred)*X[:,9] - (1-Y)*pred*X[:,9])+(lamb*w[
        gradient[11] = -1 * (sum(Y*(1-pred)*X[:,10] - (1-Y)*pred*X[:,10])+(lamb*
        gradient[12] = -1 * (sum(Y*(1-pred)*X[:,11] - (1-Y)*pred*X[:,11])+(lamb*
        gradient[13] = -1 * (sum(Y*(1-pred)*X[:,12] - (1-Y)*pred*X[:,12])+(lamb*
        gradient[14] = -1 * (sum(Y*(1-pred)*X[:,13] - (1-Y)*pred*X[:,13])+(lamb*
        gradient[15] = -1 * (sum(Y*(1-pred)*X[:,14] - (1-Y)*pred*X[:,14])+(lamb*
        gradient[16] = -1 * (sum(Y*(1-pred)*X[:,15] - (1-Y)*pred*X[:,15])+(lamb*
        gradient[17] = -1 * (sum(Y*(1-pred)*X[:,16] - (1-Y)*pred*X[:,16])+(lamb*
        gradient[18] = -1 * (sum(Y*(1-pred)*X[:,17] - (1-Y)*pred*X[:,17])+(lamb*
        gradient[19] = -1 * (sum(Y*(1-pred)*X[:,18] - (1-Y)*pred*X[:,18])+(lamb*
        gradient[20] = -1 * (sum(Y*(1-pred)*X[:,19] - (1-Y)*pred*X[:,19])+(lamb*
        gradient[21] = -1 * (sum(Y*(1-pred)*X[:,20] - (1-Y)*pred*X[:,20])+(lamb*
        gradient[22] = -1 * (sum(Y*(1-pred)*X[:,21] - (1-Y)*pred*X[:,21])+(lamb*
        gradient[23] = -1 * (sum(Y*(1-pred)*X[:,22] - (1-Y)*pred*X[:,22])+(lamb*
        gradient[24] = -1 * (sum(Y*(1-pred)*X[:,23] - (1-Y)*pred*X[:,23])+(lamb*
        gradient[25] = -1 * (sum(Y*(1-pred)*X[:,24] - (1-Y)*pred*X[:,24])+(lamb*
        gradient[26] = -1 * (sum(Y*(1-pred)*X[:,25] - (1-Y)*pred*X[:,25])+(lamb*
        gradient[27] = -1 * (sum(Y*(1-pred)*X[:,26] - (1-Y)*pred*X[:,26])+(lamb*
        return gradient
```

In [11]:
```
def update_weights(c_g, prev_weights, learning_rate, iteratnum=100, lamb=1):
    UWlist = []
    iterat = 0
    #count = 1
    while True:
        prev_weights = c_g
        w0 = prev_weights[0] - learning_rate*calculate_gradients(prev_weights,X,
        w1 = prev_weights[1] - learning_rate*calculate_gradients(prev_weights,X,
        w2 = prev_weights[2] - learning_rate*calculate_gradients(prev_weights,X,
        w3 = prev_weights[3] - learning_rate*calculate_gradients(prev_weights,X,
        w4 = prev_weights[4] - learning_rate*calculate_gradients(prev_weights,X,
        w5 = prev_weights[5] - learning_rate*calculate_gradients(prev_weights,X,
        w6 = prev_weights[6] - learning_rate*calculate_gradients(prev_weights,X,
        w7 = prev_weights[7] - learning_rate*calculate_gradients(prev_weights,X,
        w8 = prev_weights[8] - learning_rate*calculate_gradients(prev_weights,X,
        w9 = prev_weights[9] - learning_rate*calculate_gradients(prev_weights,X,
        w10 = prev_weights[10] - learning_rate*calculate_gradients(prev_weights,
        w11 = prev_weights[11] - learning_rate*calculate_gradients(prev_weights,
        w12 = prev_weights[12] - learning_rate*calculate_gradients(prev_weights,
        w13 = prev_weights[13] - learning_rate*calculate_gradients(prev_weights,
        w14 = prev_weights[14] - learning_rate*calculate_gradients(prev_weights,
        w15 = prev_weights[15] - learning_rate*calculate_gradients(prev_weights,
        w16 = prev_weights[16] - learning_rate*calculate_gradients(prev_weights,
        w17 = prev_weights[17] - learning_rate*calculate_gradients(prev_weights,
        w18 = prev_weights[18] - learning_rate*calculate_gradients(prev_weights,
        w19 = prev_weights[19] - learning_rate*calculate_gradients(prev_weights,
        w20 = prev_weights[20] - learning_rate*calculate_gradients(prev_weights,
        w21 = prev_weights[21] - learning_rate*calculate_gradients(prev_weights,
        w22 = prev_weights[22] - learning_rate*calculate_gradients(prev_weights,
        w23 = prev_weights[23] - learning_rate*calculate_gradients(prev_weights,
        w24 = prev_weights[24] - learning_rate*calculate_gradients(prev_weights,
        w25 = prev_weights[25] - learning_rate*calculate_gradients(prev_weights,
        w26 = prev_weights[26] - learning_rate*calculate_gradients(prev_weights,
        w27 = prev_weights[27] - learning_rate*calculate_gradients(prev_weights,
        c_g = [w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13, w14,
        UWlist.append(c_g)
        #count = count + 1
        #print(count)
        if (c_g[0]-prev_weights[0])**2 + (c_g[1]-prev_weights[1])**2 + (c_g[2]-p
            return c_g, UWlist

        if iterat>iteratnum:
            return c_g, UWlist
        iterat = iterat + 1
```

In [12]:
```
randomlist = []
for i in range(0,28):
    n = random.randint(1,20)
    randomlist.append(n)
print(randomlist)
```

```
[15, 20, 7, 10, 13, 4, 1, 20, 18, 5, 3, 18, 2, 5, 5, 2, 7, 1, 14, 16, 11, 16, 1
1, 18, 17, 18, 15, 16]
```

In [26]:
```
randomlist = []
for i in range(0,28):
    n = random.randint(1,20)
    randomlist.append(n)
inputweights = randomlist

# Define your functions here

def main(X, Y, w, learning_rate = 0.05, num_steps = 500, lamb=1):
    inputweights = w
    UW, UWlist = update_weights(inputweights,inputweights,learning_rate,iteratnu
    return UW, UWlist

UW, UWlist = main(X=X, Y=Y, w=inputweights, learning_rate = 0.05, num_steps = 50
```

In [14]:
```
final_weights = UW
inputvalue = X
print("These are my final weights: " + str(final_weights))
```

```
These are my final weights: [111.23248463023553, 114.32693181317917, 62.3870051
1752238, 89.63427909662757, 56.918441898351574, 147.08961124246048, 203.4649120
997185, 81.68046644879682, 129.13542147205123, 139.45075799708462, 274.30090994
3124, 60.08637243342439, 149.3138471095328, 158.23595493733797, 252.62894515938
55, -60.33365452668486, 86.5748387170658, 96.8274197157939, 83.24810620996178,
144.9474799306385, 134.27805994229794, -22.302636541139204, -115.3005003031853
2, -81.44025587819895, -83.87836611312214, -96.09036286041666, -152.23336375377
426, -78.34979969033074]
```

In [15]:
```
def finalpred(w,X):
    z = np.array(w[0]+w[1]*np.array(X[0])
                 +w[2]*np.array(X[1])+w[3]*np.array(X[2])
                 +w[4]*np.array(X[3])+w[5]*np.array(X[4])
                 +w[6]*np.array(X[5])+w[7]*np.array(X[6])
                 +w[8]*np.array(X[7])+w[9]*np.array(X[8])
                 +w[10]*np.array(X[9])+w[11]*np.array(X[10])
                 +w[12]*np.array(X[11])+w[13]*np.array(X[12])
                 +w[14]*np.array(X[13])+w[15]*np.array(X[14])
                 +w[16]*np.array(X[15])+w[17]*np.array(X[16])
                 +w[18]*np.array(X[17])+w[19]*np.array(X[18])
                 +w[20]*np.array(X[19])+w[21]*np.array(X[20])
                 +w[22]*np.array(X[21])+w[23]*np.array(X[22])
                 +w[24]*np.array(X[23])+w[25]*np.array(X[24])
                 +w[26]*np.array(X[25])+w[27]*np.array(X[26]))
    return sigmoid(z)
```

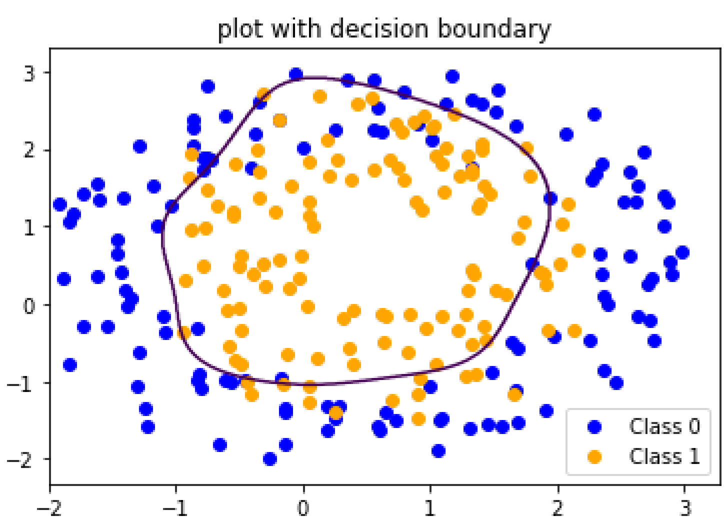
In [16]:
```
xvalue = x28D[0,:]
xvalue[27]
look = xvalue
prediction = finalpred(final_weights, xvalue)
```

In [17]:
```
datasize,DIM = x28D.shape
zmatrix = []
for value in range(datasize):
    xvalue = x28D[value,:]
    zvalue = finalpred(final_weights, xvalue)
    zmatrix.append(zvalue)
zmatrix = np.array(zmatrix)
```

In [18]:
```
x1 = np.linspace(-2, 3, 500)
x2 = np.linspace(-2, 3, 500)
x1mesh, x2mesh = np.meshgrid(x1, x2)
row,col = x1mesh.shape
zmatrix = np.zeros((row,col))
citer = 0
for r in range(row):
    for c in range(col):
        xaugment = np.array([[x1mesh[r][c],x2mesh[r][c]]])
        x28Dmesh = map_feature(xaugment, degree=6)
        xvalue = x28Dmesh[0,:]
        zvalue = finalpred(final_weights, xvalue)
        np.put(zmatrix, [citer], zvalue)
        citer = citer + 1
```

In [19]:
```
# Plot decision boundary
plt.contour(x1mesh,x2mesh,zmatrix, c=['black'], levels = [0.5])
plt.scatter(x[:126,0], x[:126,1], c = 'blue', label="Class 0")
plt.scatter(x[127:,0], x[127:,1], c = 'orange', label ="Class 1")
plt.title("plot with decision boundary")
plt.legend()
plt.show()
```

```
C:\Users\rdesa\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: UserWarnin
g: The following kwargs were not used by contour: 'c'
```



P4. Tune the strength of regularization

TODO

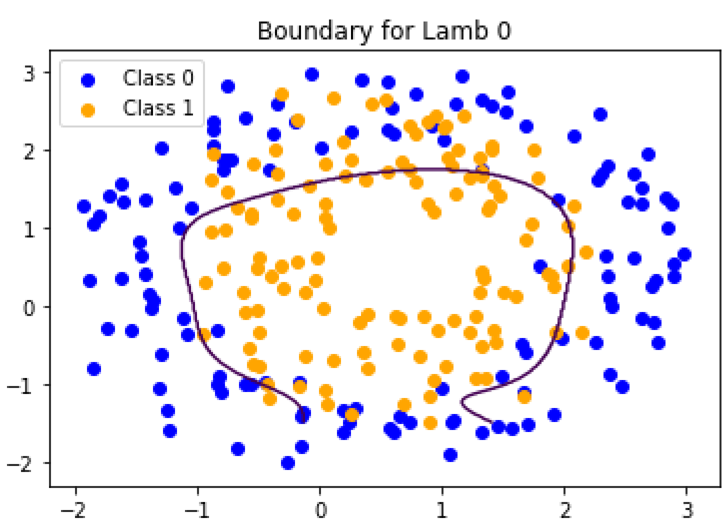
- tweak the hyper-parameter λ to be $[0, 1, 100]$
- draw the decision boundaries

In [28]:
```
x1 = np.linspace(-1.5, 2.5, 250)
x2 = np.linspace(-1.5, 2.5, 250)
x1mesh, x2mesh = np.meshgrid(x1, x2)
```

In [38]:
```python
# Lambda = 0
initallist = [15, 20, 7, 10, 13, 4, 1, 20, 18, 5, 3, 18, 2, 5, 5, 2, 7, 1, 14, 1
inputweights = initallist

UW_lamb0, UWlist = main(X=X, Y=Y, w=inputweights, learning_rate = 0.04, num_step
print("Weights for Lamb0 Complete")
final_weights_lamb0 = UW_lamb0

row,col = x1mesh.shape
zmatrix_lamb0 = np.zeros((row,col))
citer = 0
for r in range(row):
    for c in range(col):
        xaugment = np.array([[x1mesh[r][c],x2mesh[r][c]]])
        x28Dmesh = map_feature(xaugment, degree=6)
        ###
        xvalue = x28Dmesh[0,:]
        ####
        zvalue_lamb0 = finalpred(final_weights_lamb0, xvalue)
        np.put(zmatrix_lamb0, [citer], zvalue_lamb0)
        citer = citer + 1

plt.contour(x1mesh,x2mesh,zmatrix_lamb0, c=['black'], levels = [0.5])
plt.scatter(x[:126,0], x[:126,1], c = 'blue', label="Class 0")
plt.scatter(x[127:,0], x[127:,1], c = 'orange', label ="Class 1")
plt.legend()
plt.title("Boundary for Lamb 0")
plt.show()

# Lambda = 1

initallist = [15, 20, 7, 10, 13, 4, 1, 20, 18, 5, 3, 18, 2, 5, 5, 2, 7, 1, 14, 1
inputweights = initallist

UW_lamb1, UWlist = main(X=X, Y=Y, w=inputweights, learning_rate = 0.04, num_step
print("Weights for Lamb1 Complete")
final_weights_lamb1 = UW_lamb1

row,col = x1mesh.shape
zmatrix_lamb1 = np.zeros((row,col))
citer = 0
for r in range(row):
    for c in range(col):
        xaugment = np.array([[x1mesh[r][c],x2mesh[r][c]]])
        x28Dmesh = map_feature(xaugment, degree=6)
        ###
        xvalue = x28Dmesh[0,:]
        ####
        zvalue_lamb1 = finalpred(final_weights_lamb1, xvalue)
        np.put(zmatrix_lamb1, [citer], zvalue_lamb1)
        citer = citer + 1

plt.contour(x1mesh,x2mesh,zmatrix_lamb1, c=['black'], levels = [0.5])
plt.scatter(x[:126,0], x[:126,1], c = 'blue', label="Class 0")
plt.scatter(x[127:,0], x[127:,1], c = 'orange', label ="Class 1")
plt.legend()
```

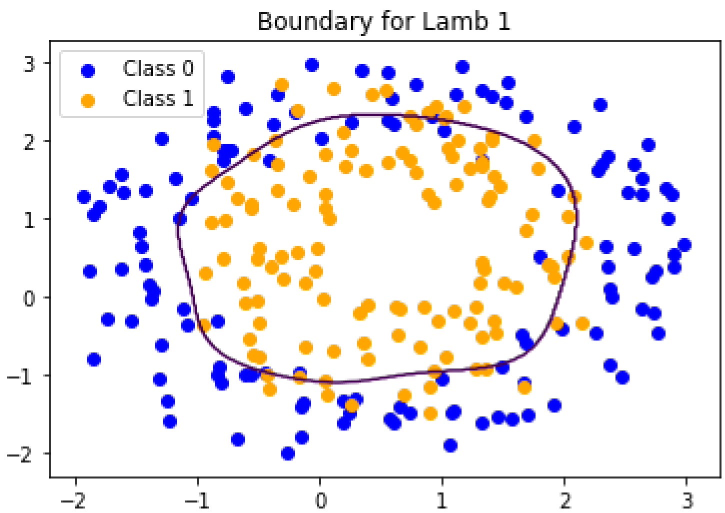
```
plt.title("Boundary for Lamb 1")
plt.show()

# Lambda = 100

initallist = [15, 20, 7, 10, 13, 4, 1, 20, 18, 5, 3, 18, 2, 5, 5, 2, 7, 1, 14, 1(
inputweights = initallist

UW_lamb100, UWlist = main(X=X, Y=Y, w=inputweights, learning_rate = 0.04, num_st(
print("Weights for Lamb 100 Complete")
final_weights_lamb100 = UW_lamb100

row,col = x1mesh.shape
zmatrix_lamb100 = np.zeros((row,col))
citer = 0
for r in range(row):
    #print("Entering Row: " + str(r))
    for c in range(col):
        xaugment = np.array([[x1mesh[r][c],x2mesh[r][c]]])
        x28Dmesh = map_feature(xaugment, degree=6)
        ###
        xvalue = x28Dmesh[0,:]
        ####
        zvalue_lamb100 = finalpred(final_weights_lamb100, xvalue)
        np.put(zmatrix_lamb100, [citer], zvalue_lamb100)
        citer = citer + 1

plt.contour(x1mesh,x2mesh,zmatrix_lamb100, c=['black'], levels = [0.5])
plt.scatter(x[:126,0], x[:126,1], c = 'blue', label="Class 0")
plt.scatter(x[127:,0], x[127:,1], c = 'orange', label ="Class 1")
plt.legend()
plt.title("Boundary for Lamb 100")
plt.show()
```

```
Weights for Lamb0 Complete

C:\Users\rdesa\Anaconda3\lib\site-packages\ipykernel_launcher.py:23: UserWarnin
g: The following kwargs were not used by contour: 'c'
```



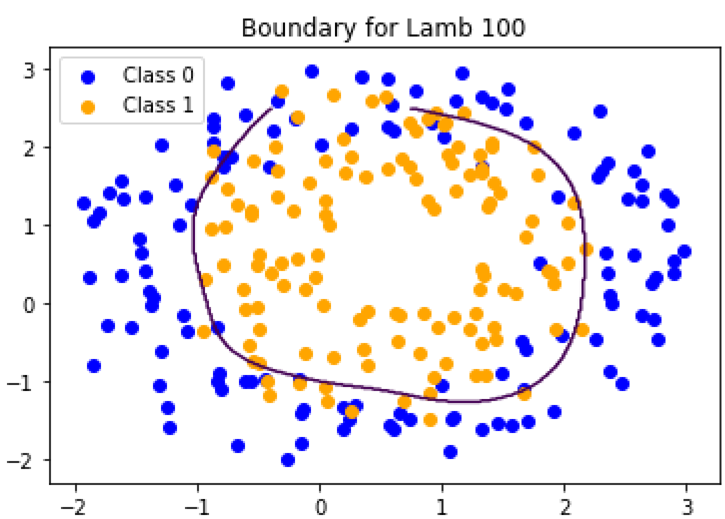

```
Weights for Lamb1 Complete

C:\Users\rdesa\Anaconda3\lib\site-packages\ipykernel_launcher.py:53: UserWarn
```

```
ing: The following kwargs were not used by contour: 'c'
```




```
Weights for Lamb 100 Complete
```

```
C:\Users\rdesa\Anaconda3\lib\site-packages\ipykernel_launcher.py:84: UserWarnin
g: The following kwargs were not used by contour: 'c'
```



Answer for part (d) here:

In [ ]:

```
#For lambda 0,1, and 100 the plots looks
#different for each graph have different
#final weights. At lambda = 0 the model
#is overfitted because weights that are
#unnessary are not being eliminated. The
#plot is trying to fit too many data
#points causing the graph to be smaller
#then nessary and open at the bottom
#At lambda = 1 the model is about right
#as the weights that are unnessary are
#being eliminated but weights that are
#important are still being maintained.
#At lambda = 100 the model is underfitted
#as weights that are required to properly
#fit the model are being minimized. The
#plot is missing data point that should be
#incompassed by the decision boundary as
#the decision boundary gets larger
```