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 [1]:
```
# load packages
import numpy as np
import pandas as pd
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 [2]:
```
# Use pandas to load q1_train.csv and q1_test.csv
# Each data point has 200 features(X), followed by 1 label(Y)

#### TODO ####
total_data = pd.read_csv("/home/akshay/Downloads/MAIL/Fall 2021 Assignment 3/q1_train.csv")
trainX = total_data.iloc[:,1:201]
trainY = total_data.iloc[:,201]
total_data = pd.read_csv("/home/akshay/Downloads/MAIL/Fall 2021 Assignment 3/q1_test.csv")
testX = total_data.iloc[:,1:201]
testY = total_data.iloc[:,201]
##############

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

In [4]:
```
# print(type(testX))
# N_train = trainX.shape[0]
# py0 = (trainY == 0).sum()/N_train
# py1 = (trainY == 1).sum()/N_train
# print(py0+py1)
trainX = trainX.to_numpy()
trainY = trainY.to_numpy()
testX = testX.to_numpy()
testY = testY.to_numpy()
print(trainX.shape)
#trainX[idx].shape
mean0 = np.mean(trainX[np.where(trainY == 0)],axis=0)
var0 = np.var(trainX[np.where(trainY == 0)],axis=0)
mean1 = np.mean(trainX[np.where(trainY == 1)],axis=0)
var1 = np.var(trainX[np.where(trainY == 1)],axis=0)
P_x_y0 = (1/(np.sqrt(2*np.pi*var0)))*\
        np.exp(-0.5*np.power((trainX-mean0)/np.sqrt(var0),2))
P_x_y1 = (1/(np.sqrt(2*np.pi*var1)))*\
        np.exp(-0.5*np.power((trainX-mean1)/np.sqrt(var1),2))
P_x_y0 = np.product(P_x_y0,axis=1)
P_x_y0.shape
```

```
(134000, 200)
```

Out[4]:
```
(134000,)
```

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 (N_train, M), where N_train is number of train samples and 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 [9]:
```
def myNBSolver(trainX, trainY, testX, testY):

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

    #### TODO ####
    # Compute P(y == 0) and P(y == 1)

    py0 = np.sum(trainY == 0)/N_train
    py1 = np.sum(trainY == 1)/N_train
    py0_test = np.sum(testY == 0)/N_test
    py1_test = np.sum(testY == 1)/N_test
    ##############
    print("Total probablity is %.2f. Should be equal to 1." %(py0 + py1))

    #### TODO ####
    # Compute mean/var for each label
    mean0 = np.mean(trainX[np.where(trainY == 0)],axis=0)
    mean1 = np.mean(trainX[np.where(trainY == 1)],axis=0)
    var0 = np.var(trainX[np.where(trainY == 0)],axis=0)
    var1 = np.var(trainX[np.where(trainY == 1)],axis=0)

    ##############
    assert(mean0.shape[0] == M)

    #### 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 feel free to
    # print something (like step number) to check the progress
    # P(xi|y == 0)
    P_x_y0 = (1/(np.sqrt(2*np.pi*var0)))*\
             np.exp(-0.5*np.power((trainX-mean0)/np.sqrt(var0),2))
    P_x_y0 = np.product(P_x_y0,axis=1)
    # P(xi|y == 0)
    P_x_y1 = (1/(np.sqrt(2*np.pi*var1)))*\
             np.exp(-0.5*np.power((trainX-mean1)/np.sqrt(var1),2))
    P_x_y1 = np.product(P_x_y1,axis=1)
    # P(xi|y == 0) for test
    P_x_y0_test = (1/(np.sqrt(2*np.pi*var0)))*\
             np.exp(-0.5*np.power((testX-mean0)/np.sqrt(var0),2))
    P_x_y0_test = np.product(P_x_y0_test,axis=1)
    # P(xi|y == 0) for test
    P_x_y1_test = (1/(np.sqrt(2*np.pi*var1)))*\
             np.exp(-0.5*np.power((testX-mean1)/np.sqrt(var1),2))
    P_x_y1_test = np.product(P_x_y1_test,axis=1)

    #calculating P(y=0/x) and P(y=1/x), we are calculating only the numerator
    # as to compare, the denominator is constant for both.
    P_x_y0 = P_x_y0*py0
    P_x_y1 *= py1
    P_x_y0_test *= py0_test
    P_x_y1_test *= py1_test

    train_pred = np.empty((N_train))
    train_pred = np.where(P_x_y0>P_x_y1,0,1)
    test_pred = np.empty((N_test))
    test_pred = np.where(P_x_y0_test>P_x_y1_test,0,1)
    ##############
    assert(train_pred[0] == 0 or train_pred[0] == 1)
    assert(test_pred[0] == 0 or test_pred[0] == 1)

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

    train_acc = np.sum(trainY == train_pred)/N_train
    test_acc = np.sum(testY == test_pred)/N_test

    ##############

    return train_acc, test_acc
```

In [10]:

```
# 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))
```

```
Total probablity is 1.00. Should be equal to 1.
```

```
Train accuracy is 92.22
Test accuracy is 92.08
```

P3. Test your result using sklearn

TODO

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

In [57]:
```
def skNBSolver(trainX, trainY, testX, testY):

    #### TODO ####
    # fit model
    # make prediction
    # compute accuracy
    model = GaussianNB()
    trained = model.fit(trainX,trainY)
    sk_train_acc = np.sum(trainY == trained.predict(trainX))/trainX.shape[0]
    sk_test_acc = np.sum(testY == trained.predict(testX))/testX.shape[0]

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

In [58]:
```
# 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
```

In [ ]:
```
```

Note for question2

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

In [41]:
```
# Import modules
from sklearn.linear_model import LinearRegression, Lasso, Ridge
import numpy as np
import matplotlib.pyplot as plt
```

P1. Create data and plot

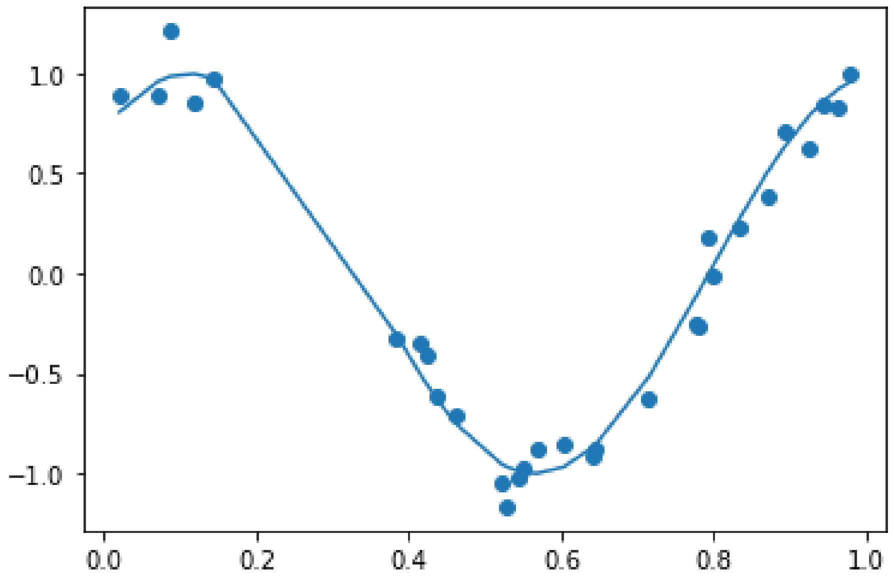
TODO

- 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 [42]:
```
# Define the function to generate data points
def generate_points(input_points):
    output_points = np.sin(2.2*np.pi*input_points + 0.8)
    return output_points

# Initialize random seed
np.random.seed(0)
input_points = np.random.uniform(0,1,size=(30,1))
input_points = np.sort(input_points,axis=0)
output_points = generate_points(input_points)

# Generate noisy data points: (x,y)
noise = np.random.normal(loc=0,scale=0.1,size=(30,1))
output_points_noise = output_points + noise
```

```
# Plot true model and sampled data points
plt.plot(input_points,output_points)
plt.scatter(input_points,output_points_noise)
```

Out[42]:
```
<matplotlib.collections.PathCollection at 0x7f1cc621e6a0>
```



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

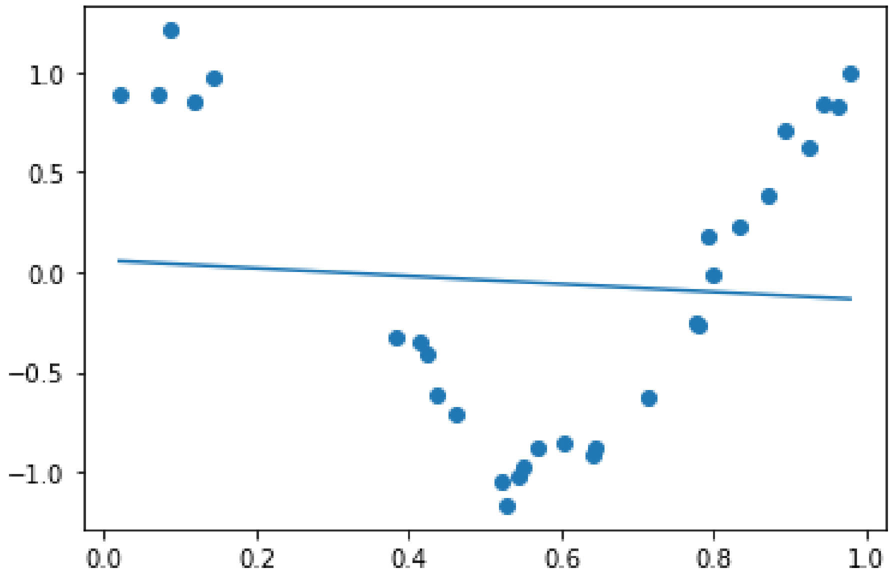
In [43]:
```
# Fit a linear model in the original space
linear_model = LinearRegression()
linear_model.fit(input_points,output_points_noise)
w0, w1 = linear_model.intercept_, linear_model.coef_
print("w0: ",w0,"w1: ",w1)

# Plot fitted linear model
predicted = linear_model.predict(input_points)
```

```
plt.plot(input_points,predicted)
plt.scatter(input_points,output_points_noise)
```

```
w0:  [0.06038094] w1:  [[-0.19787027]]
```

Out[43]:

```
<matplotlib.collections.PathCollection at 0x7f1cc61fc130>
```



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 [44]:

```
# Augment the original feature to a 15-vector
def total_feature_generate(input_points):
    total_feature = input_points
    for i in range(14):
        new_feature = np.power(input_points,i+2)
        total_feature = np.concatenate((total_feature,new_feature),axis=1)
    return total_feature
```

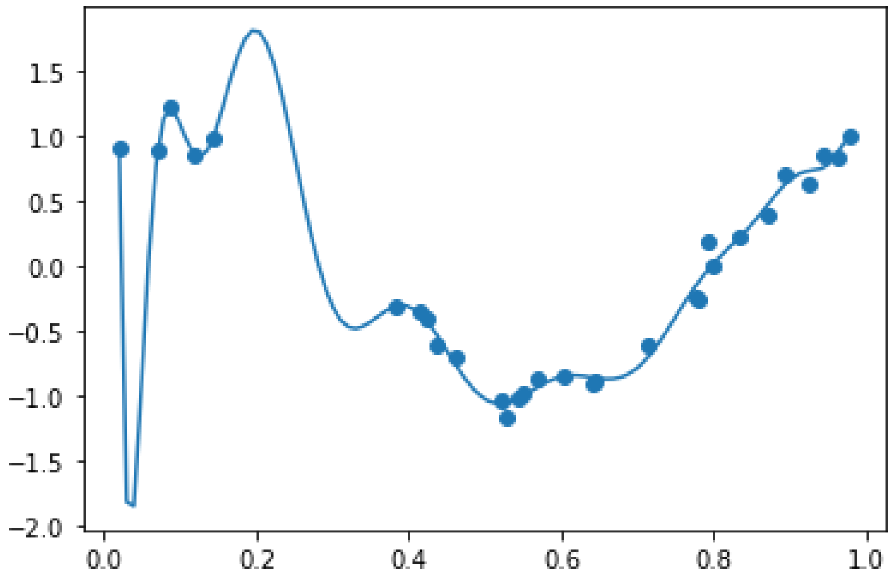
In [45]:

```
# Fit linear model to the generated 15-vector features
total_feature = total_feature_generate(input_points)
poly_linear_model = LinearRegression()
poly_linear_model.fit(total_feature,output_points_noise)
w = np.zeros(16)

w[0] = poly_linear_model.intercept_
w[1:] = poly_linear_model.coef_
print("Weights, the first term is the bias: \n",w)
print("Score:",poly_linear_model.score(total_feature,output_points_noise))

predicted = poly_linear_model.predict(total_feature)

# Plot fitted curve and sampled data points
x = np.linspace(np.min(input_points),np.max(input_points),100)
x = x.reshape(-1,1)
#print(x)
x_total_feature = total_feature_generate(x)
y = poly_linear_model.predict(x_total_feature)
#print(x)
#plt.plot(input_points,predicted)
plt.plot(x,y)
plt.scatter(input_points,output_points_noise)
```

```
Weights, the first term is the bias:
 [ 3.11668855e+01 -2.97811934e+03  1.03893518e+05 -1.87420290e+06
  2.03717119e+07 -1.44873537e+08  7.09317173e+08 -2.47066536e+09
  6.24563083e+09 -1.15676914e+10  1.56895523e+10 -1.54006637e+10
  1.06457712e+10 -4.91379706e+09  1.35920276e+09 -1.70381611e+08]
Score: 0.9914608458991773
```

Out[45]:

```
<matplotlib.collections.PathCollection at 0x7f1cc61509d0>
```



P4. Lasso regularization

TODO

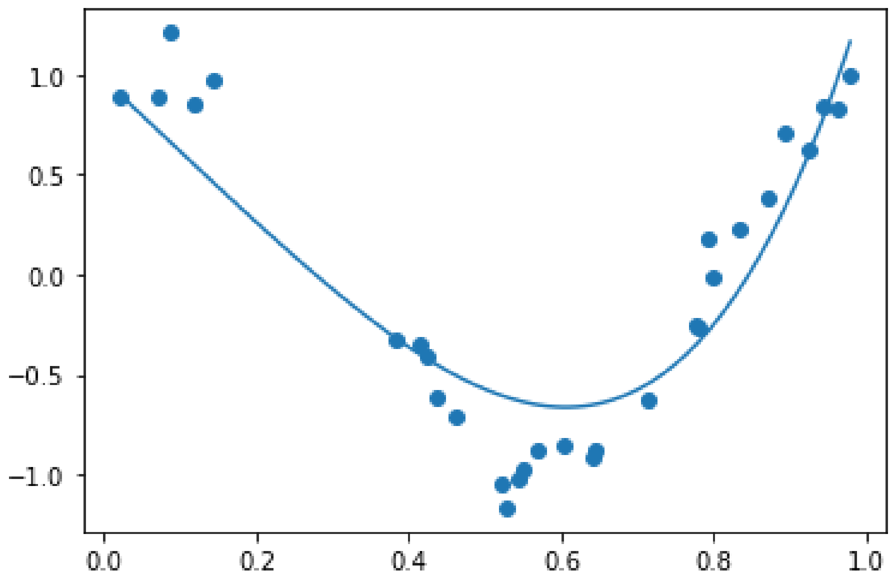
- use sklearn to fit a 15-degree polynomial model with L1 regularization
- report w
- plot the fitted model $h(x)$ together with data points

In [46]:

```
# Fit 15-degree polynomial with L1 regularization
# Start with lambda(alpha) = 0.01 and max_iter = 1e4
lasso_regression = Lasso(alpha=.01,max_iter=1e4)
# Plot fitted curve and sampled data points
lasso_regression.fit(total_feature,output_points_noise)
#w holds the weights
w = np.zeros(16)
w[0] = lasso_regression.intercept_
w[1:] = lasso_regression.coef_
print(w,"\nScore",lasso_regression.score(total_feature,output_points_noise))
predicted = lasso_regression.predict(x_total_feature)

plt.plot(x,predicted)
plt.scatter(input_points,output_points_noise)
```

```
[ 0.97968677 -3.61175387  0.          0.          4.05737553  0.
  0.          0.          0.          0.          0.          0.
  0.         -0.         -0.         -0.        ]
Score 0.8642604684665177
```

```
Out[46]: <matplotlib.collections.PathCollection at 0x7f1cc6129e20>
```

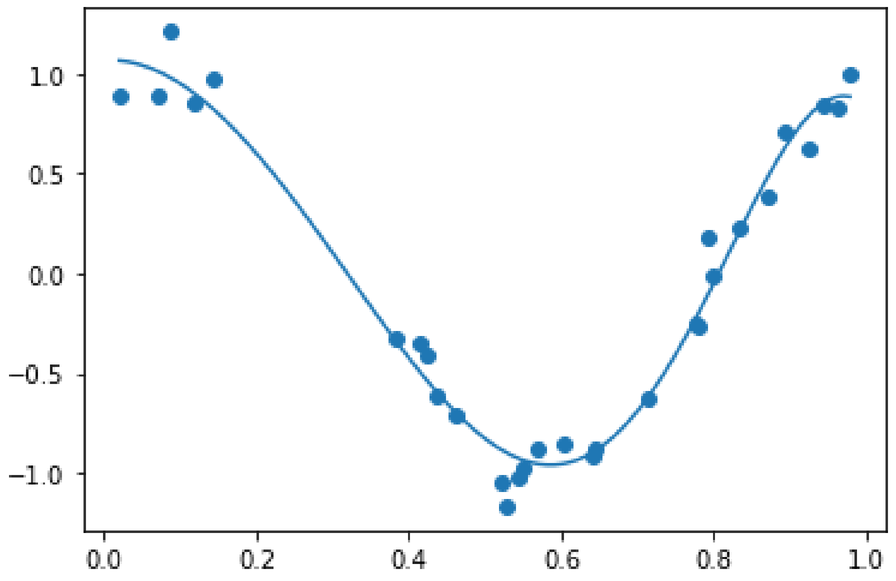


```
In [47]:
lasso_regression = Lasso(alpha=.0001,max_iter=1e4)
# Plot fitted curve and sampled data points
lasso_regression.fit(total_feature,output_points_noise)
#w holds the weights
w = np.zeros(16)
w[0] = lasso_regression.intercept_
w[1:] = lasso_regression.coef_
print(w,"\nScore",lasso_regression.score(total_feature,output_points_noise))
predicted = lasso_regression.predict(x_total_feature)

plt.plot(x,predicted)
plt.scatter(input_points,output_points_noise)
```

```
[  1.06843622   0.2167843  -14.45629376   5.73351697  14.49985714
   0.           0.          -0.          -0.          -5.1753093
  -2.49636651  -0.          -0.          -0.           0.
   1.45189882]
Score 0.9747580523845099
```

```
/home/akshay/anaconda3/lib/python3.8/site-packages/sklearn/linear_model/_coordinate_descent.py:53
0: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterat
ions. Duality gap: 0.02063625723763507, tolerance: 0.0017515538543778103
  model = cd_fast.enet_coordinate_descent(
```

Out[47]:
```
<matplotlib.collections.PathCollection at 0x7f1cc6081b80>
```



Reporting best lambda and weights

In [48]:
```python
print("weights: The first value is the intercept:\n",w)
print("lambda = .0001")
```

```
weights: The first value is the intercept:
 [  1.06843622   0.2167843  -14.45629376   5.73351697  14.49985714
   0.           0.          -0.           -0.          -5.1753093
  -2.49636651  -0.          -0.           -0.           0.
   1.45189882]
lambda = .0001
```

Observations

As the value of lambda decreases the value the number of non-zero elements in the weights decreases. For example lambda = 0.0001 has 8 non-zero values as compared to 4 non zero values when lambda=.01. This means increasing lambda tries to underfit the data. **Increasing lambda decreases the weights**

Understanding of lasso

As any regularization method lasso also tries to reduce overfitting.The a proportion(ie lambda times) magnitudes of the weights are subtracted from the weights in the training process. Lasso can lead to zero coefficients i.e. some of the features are completely neglected for the evaluation of output. This means that lasso can also help in completely removing some features which cause overfitting by making their weights 0.

Tweaking lambda

So when lambda is decreases, overfitting increases. I have displayed the scores for two different values of lambda. When lambda is very high the model is very underfitted.

P5. Ridge regularization

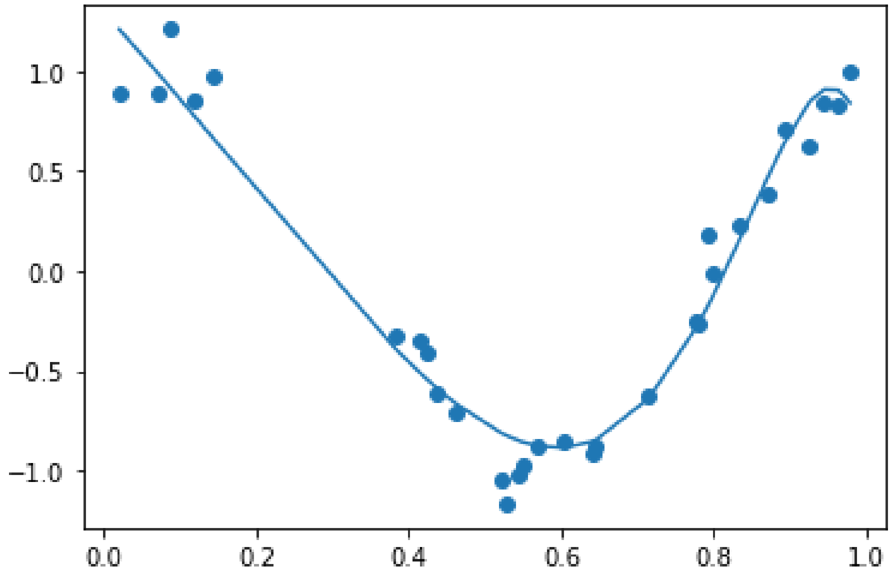
TODO

- 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 [49]:
```
# Fit 15-degree polynomial with L2 regularization
# Start with lambda(alpha) = 0.01 and max_iter = 1e4
ridge_regression_model = Ridge(alpha=0.01,max_iter=1e4)
ridge_regression_model.fit(total_feature,output_points_noise)

predicted = ridge_regression_model.predict(total_feature)
w = np.zeros(16)
w[0] = ridge_regression_model.intercept_
w[1:] = ridge_regression_model.coef_
plt.plot(input_points,predicted)
plt.scatter(input_points,output_points_noise)
```

```
# Plot fitted curve and sampled data points and compare to L1 regularization from P4
print("weights: (The first value is the intercept):\n",w)
print(ridge_regression_model.score(total_feature,output_points_noise))
```

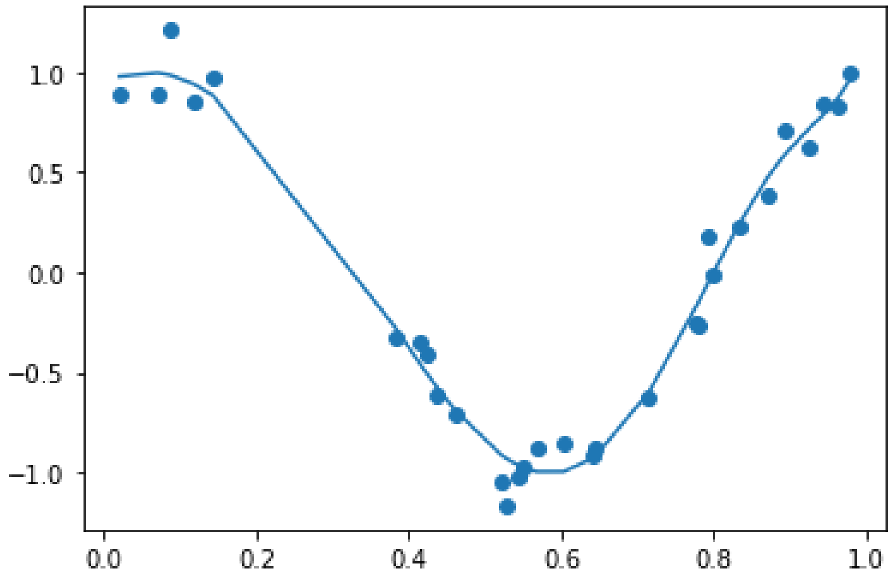
```
weights: (The first value is the intercept):
 [ 1.297752   -4.21231562 -1.58994435  1.38570425  2.48459438  2.39535518
  1.78896615  1.05511527  0.38500882 -0.14394475 -0.51380601 -0.73493061
 -0.82911584 -0.82098452 -0.73390704 -0.58833042]
0.9533749611502453
```



The best value of lambda found is .0001

In [50]:
```
ridge_regression_model = Ridge(alpha=0.0001,max_iter=1e4)
ridge_regression_model.fit(total_feature,output_points_noise)

predicted = ridge_regression_model.predict(total_feature)
w = np.zeros(16)
w[0] = ridge_regression_model.intercept_
w[1:] = ridge_regression_model.coef_
plt.plot(input_points,predicted)
plt.scatter(input_points,output_points_noise)
# Plot fitted curve and sampled data points and compare to L1 regularization from P4
print("weights: (The first value is the intercept):\n",w)
print(ridge_regression_model.score(total_feature,output_points_noise))
```

```
weights: (The first value is the intercept):
 [  0.94933395   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]
0.9804373188863559
```



Note for question3

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

In [2]:

```python
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

P1. Load data and plot

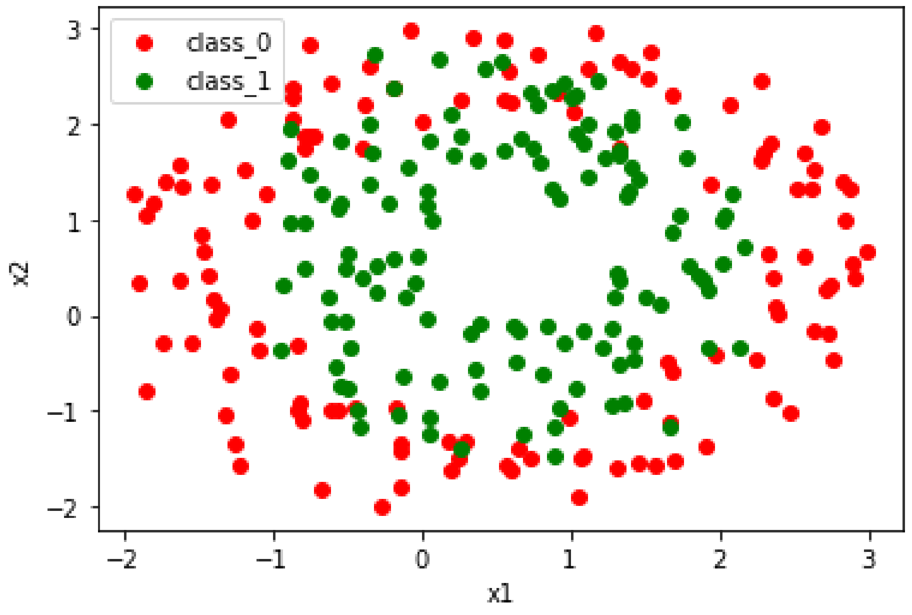
TODO

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

In [36]:

```python
# Load dataset
data = pd.read_csv("q3_data.csv")
data = data.to_numpy()

# Plot points
class_1_idx = np.where(data[:,2] == 1)
class_1 = data[class_1_idx,:2].squeeze()
class_0 = data[np.where(data[:,2] == 0),:2].squeeze()

fig, ax = plt.subplots(1)
ax.scatter(class_0[:,0], class_0[:,1], color='r', label="class_0")
ax.scatter(class_1[:,0], class_1[:,1], color='g', label="class_1")
ax.set_xlabel("x1")
ax.set_ylabel("x2")
ax.legend()
print(class_1.shape, class_0.shape)
```

```
(126, 2) (125, 2)
```



P2. Feature mapping

TODO

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

In [4]:

```
# Transform points to 28D space
def map_feature(data):
    #taking the first column of input ie x1 and x2
    x1 = data[:,0]
    x1 = np.expand_dims(x1,axis=1)
    x2 = data[:,1]
    x2 = np.expand_dims(x2,axis=1)

    curr = np.concatenate((x1,x2),axis=1)
    #created an array named final, whose first column is 1
    final = np.full((x1.shape[0],1),1)
    #concatenate 1,x1,x2 array
    final = np.concatenate((final,curr),axis=1)

    for i in range(2,7,1):
        m = int(np.floor(curr.shape[1]/2)+1)
        n = int(np.floor(curr.shape[1]/2))
        x1_part = curr[:,0:m]*x1
        x2_part = curr[:,n:curr.shape[1]]*x2
        #make the new feature matrix columns by multiplying previous generated features
        #with x1 and x2
        curr = np.concatenate((x1_part,x2_part),axis=1)
        #append the currently generated array to the final
        final = np.concatenate((final,curr),axis=1)
    return final

final = map_feature(data)
```

In [5]:

```
# Define your functions here
def sigmoid(x):
    #returns the sigmoid of input data
     sig_x = 1/(1 + np.exp(-x))
     return sig_x

def calculate_gradients(Y,X,sig_x,lambd,updated_weights):
    #calculating the gradients
    grad = X*(sig_x-Y)

    #defining regularization functions
    regularization = updated_weights.copy()
    #implementing regularization only for weights except bias
    regularization[0] = 0
    regularization[1:] *= lambd

    loss = -Y * np.log10(sig_x) -(1-Y)*np.log10(1-sig_x)
    loss = np.sum(loss,axis=0)
    loss_regularization = np.sum(np.power(updated_weights,2),axis=0)/(2*Y.shape[0])
    loss += loss_regularization

    global loss_list
    loss_list.append(loss)

    current_grads = np.sum(grad,axis=0)
    current_grads = np.expand_dims(current_grads,axis=1)
    #calculating gradients using L2 regularization
    current_grads = (current_grads+regularization)/Y.shape[0]
    return current_grads

def update_weights(prev_weights, current_grads, learning_rate):
    #updating the weights
    prev_weights -= learning_rate*current_grads
    return prev_weights

def main(X, Y, weights,alpha,learning_rate, num_steps):
    updated_weights = weights

    for j in range(num_steps):
        sig_x = sigmoid(X@updated_weights)
        predicted = np.where(sig_x<0.5,0,1)
        accuracy = np.sum(predicted == Y)/Y.shape[0]
        #print(accuracy)
        #print(sig_x.shape,X.shape,Y.shape)
        current_grads = calculate_gradients(Y,X,sig_x,alpha,updated_weights)
        #current_grads = np.expand_dims(current_grads,axis=1)
        updated_weights = update_weights(updated_weights,current_grads,learning_rate)

    return updated_weights

def predict(final_weights,X):
    sig_x = sigmoid(X@final_weights)
    return sig_x
```

In [6]:

```
def logistic_regression_regularized(alpha,initial_weights,lr,number_of_iterations):
    X = final
    Y = data[:,2]
    Y = np.expand_dims(Y,axis=1)
    final_weights = main(X,Y,weights=initial_weights,alpha=alpha,learning_rate=lr,num_steps=numbe
    #print(final_weights.shape,X.shape)
    predicted = predict(final_weights,X)
    predicted = np.where(predicted<0.5,0,1)
    accuracy = np.sum(predicted == Y)/Y.shape[0]
    print("Accuracy for lambda=",alpha," : ",accuracy)#,"Coefficients: ",final_weights)
    return final_weights
```

In [37]:

```
# Plot decision boundary
```

```
nx, ny = (3, 2)
x = np.linspace(0, 1, nx)
y = np.linspace(0, 1, ny)
xv, yv = np.meshgrid(x, y)
x,y,xv, yv
h = .01
x_min, x_max = data[:, 0].min(), data[:, 0].max()
y_min, y_max = data[:, 1].min(), data[:, 1].max()
x1, y1 = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

xx = x1.reshape(-1,1)
yy = y1.reshape(-1,1)
mesh_data = np.concatenate([xx,yy],axis = 1)
mesh_data = map_feature(mesh_data)

def plot_countour(final_weights):
    predicted_mesh = predict(final_weights,mesh_data)
    predicted_mesh = np.where(predicted_mesh<0.5,0,1)
    predicted_mesh = predicted_mesh.reshape(x1.shape)

    fig, ax = plt.subplots(1)
    ax.scatter(class_0[:,0], class_0[:,1], color='r', label="class_0")
    ax.scatter(class_1[:,0], class_1[:,1], color='g', label="class_1")
    ax.set_xlabel("x1")
    ax.set_ylabel("x2")
    ax.legend()
    ax.contour(x1,y1,predicted_mesh,levels=[1])
```

P4. Tune the strength of regularization

TODO

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

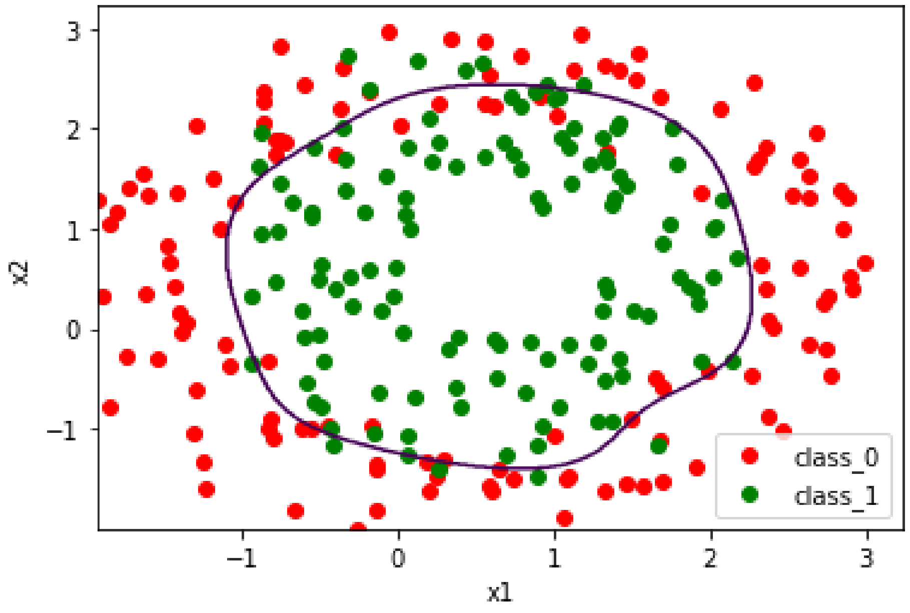
In [38]:

```
# lambda = 0
loss_list = []
```

```
initial_weight = np.zeros((28,1))
lr = .005
number_of_iterations = 65000
final_weights = logistic_regression_regularized(0,initial_weight,lr,number_of_iterations)
print("final weights: ",final_weights[:,0])
plot_countour(final_weights)
iteration_list = np.arange(1,number_of_iterations+1,1)
```

```
Accuracy for lambda= 0  :  0.8685258964143426
final weights:  [ 3.41421127  0.09472157  0.63773323 -0.30190954 -0.9114259  -0.12601866
  0.28713364  1.04951738 -0.32212943 -1.01455583 -1.30320873  0.20956718
 -0.53474073 -0.01875392 -0.3928393   1.26532246  0.57789427  0.35449801
  0.41480507  1.21375678  0.87640593 -0.34797747 -0.3189492  -0.15930069
 -0.19518332 -0.2517759  -0.38654135 -0.25222523]
```

```
<ipython-input-37-f12fe0a94fe7>:29: UserWarning: No contour levels were found within the data rang
e.
  ax.contour(x1,y1,predicted_mesh,levels=[1])
```



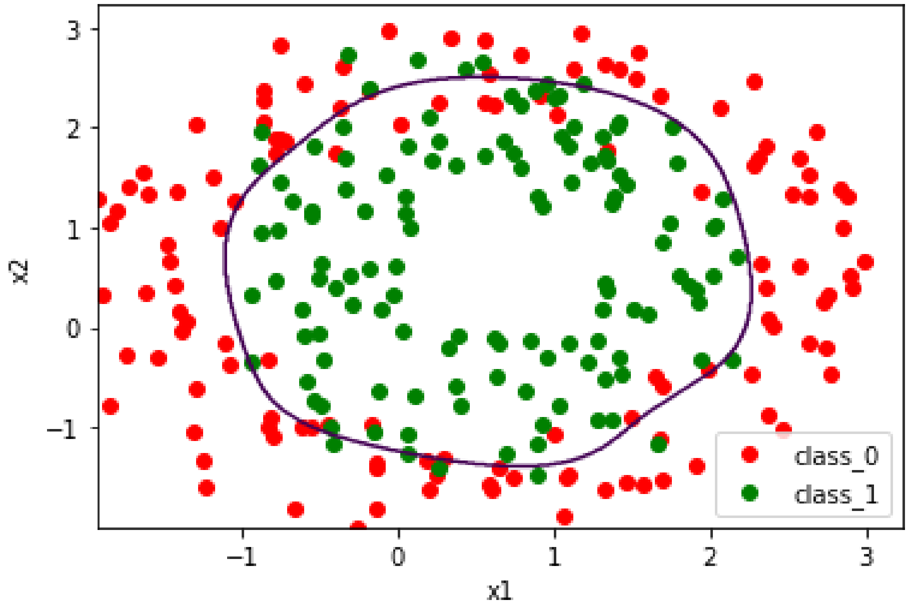
In [39]:

```
# lambda = 1
loss_list = []
lr = .005
number_of_iterations = 65000
initial_weight = np.zeros((28,1))
final_weights = logistic_regression_regularized(1,initial_weight,lr,number_of_iterations)
```

```
plot_countour(final_weights)
print("final weights: ",final_weights[:,0])
```

```
Accuracy for lambda= 1  :  0.8725099601593626
final weights:  [ 3.19875908  0.2166378   0.47505822 -0.43078569 -0.53243825 -0.28807149
  0.28434898  0.73949422 -0.0463839  -0.50945156 -0.87360264  0.1679833
 -0.3250541   0.02457974 -0.26620888  0.88296733  0.44149728  0.24470545
  0.30527777  0.92957401  0.64501353 -0.25933276 -0.23408823 -0.16372955
 -0.14263605 -0.22224787 -0.30758602 -0.19964328]
```

```
<ipython-input-37-f12fe0a94fe7>:29: UserWarning: No contour levels were found within the data rang
e.
  ax.contour(x1,y1,predicted_mesh,levels=[1])
```

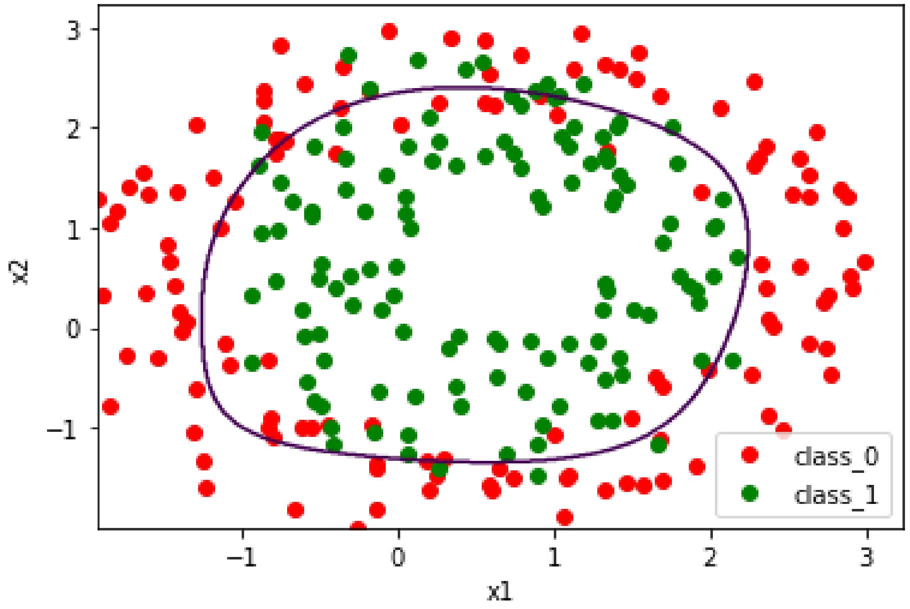


In [40]:

```
# lambda = 100
loss_list = []
lr = .005
number_of_iterations = 65000
initial_weight = np.zeros((28,1))
final_weights = logistic_regression_regularized(100,initial_weight,lr,number_of_iterations)
plot_countour(final_weights)
print("final weights: ",final_weights[:,0])
```

```
Accuracy for lambda= 100  :  0.8207171314741036
final weights:  [ 1.61749735  0.10032936  0.07711003 -0.03337203  0.02790076 -0.00267251
  0.13152594  0.05460935  0.10774059  0.11905399 -0.05316493  0.05228614
  -0.01311043  0.09879912  0.04485398  0.20034653  0.0410622   0.06245469
   0.00831589  0.16565254  0.21172377 -0.1138199   0.01219723 -0.09680202
   0.00397365 -0.16027304 -0.04592887 -0.11870196]
```

```
<ipython-input-37-f12fe0a94fe7>:29: UserWarning: No contour levels were found within the data rang
e.
  ax.contour(x1,y1,predicted_mesh,levels=[1])
```



Answer for part (d) here:

But increasing lambda decreases the over fitting, and also it brings down the value of coefficients of the weights of the model and makes it close to 0. So for lambda = 100, the accuracy very lower compared to lambda=0,1, as well as the weights have a lower magnitutude. It shows the data is underfitted. So as the lambda increases overfitting reduces