Logistic Regression

CS 4650 "Natural Language Processing" - Project 0
Georgia Tech, Spring 2023

(Instructor: Prof. Wei Xu; TAs: Ben Podrazhansky, Marcus Ma, Mounica Maddela, Rahul Katre)

In this assignment, we will walk you through the process of implementing logistic regression from scratch. You will also apply your implemented logistic regression model to a small dataset and predict whether a student will be admitted to a university. This dataset will allow you to visualize the data and debug more easily. You may find **this documentation** very helpful, though it is about how to implement logistic regression in Octave, other than Python.

This assignment also serves as a programming preparation test. We will use Numpy -- a popular Python package for scientific computing and implementing machine learning algorithms. It provides very good support for matrix and vector operations. You need to feel comfortable working with matrics, vectors, and tensors in order to complete all the programming projects in CS 650.

IMPORTANT: In this assignment, except Numpy and Matplotlib, no other external Python packages are allowed. Scipy package can be used in gradient checking, though, it is not allowed elsewhere.

0. Honor Code [1 points]

Honor Code: I hereby agree to abide the Georgia Tech's Academic Honor Code, promise that the submitted assignment is my own work, and understand that my code is subject to plagiarism test.

Signature: *Yueqiao Chen*

1. Importing Numpy and Matplotlib [Code provided - do not change]

```
import sys

# Check what version of Python is running
print (sys.version)

    3.8.10 (default, Nov 14 2022, 12:59:47)
    [GCC 9.4.0]
```

We will also import Matplotlib, a Python package for data visualization.

```
# Run some setup code for this notebook. Don't modify anything in this cell.

import random
import numpy as np
import matplotlib.pyplot as plt

# This is a bit of magic to make matplotlib figures appear inline in the notebook
# rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# reload external python modules;
# http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

Show hidden output

2. Visualizing the Data [Code provided - no need to change]

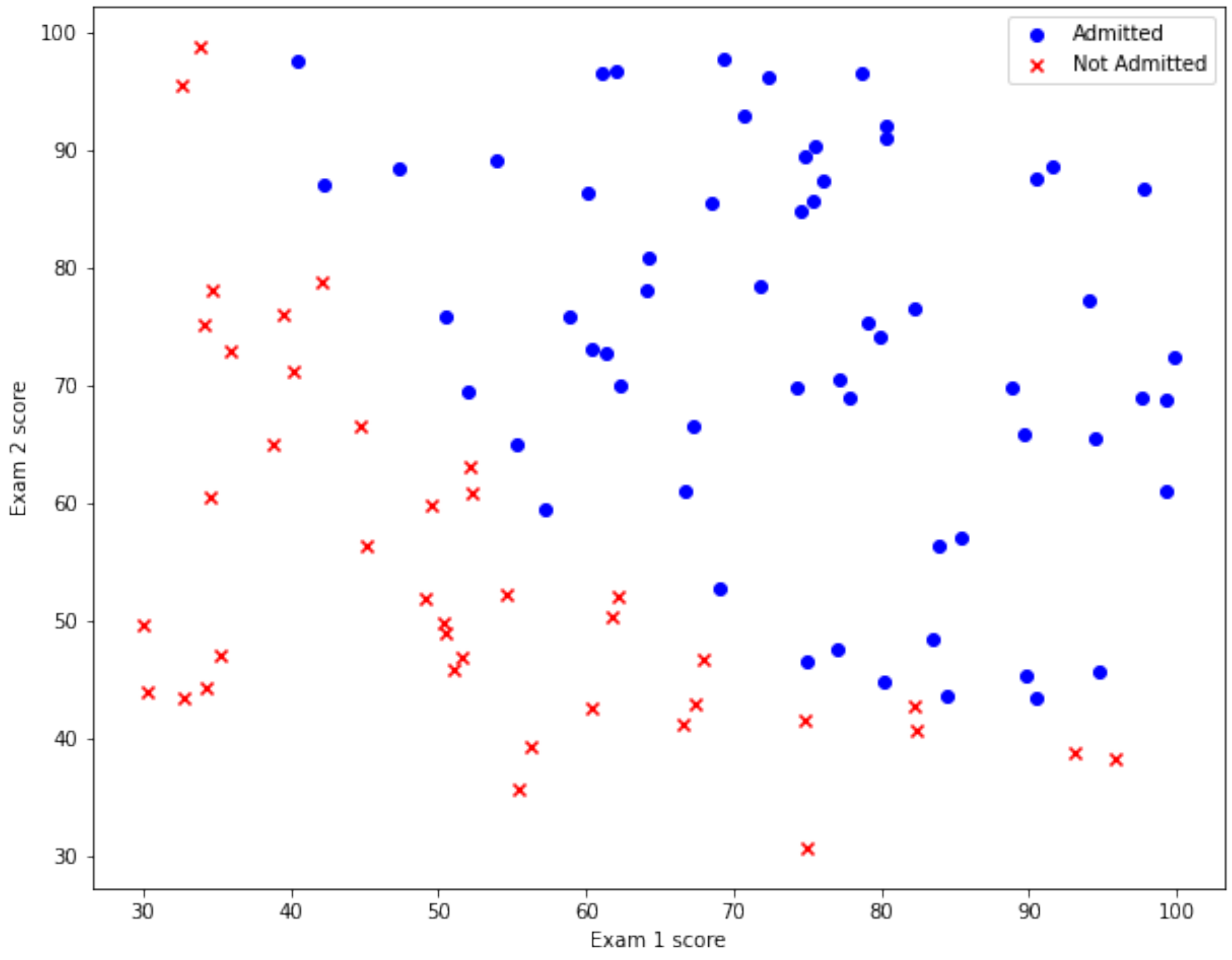
The provided dataset contains applicants' scores on two exams and the admission decisons for 100 students. This dataset will allow us to visualize in a 2D figure and showcase how the logistic regression algorithm works more intuitively.

```
!wget https://raw.githubusercontent.com/cocoxu/CS4650_projects_sp2022/master/p0/p0_
```

```
    --2023-01-26 22:54:23--  https://raw.githubusercontent.com/cocoxu/CS4650_proje
    Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108
    Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.10
    HTTP request sent, awaiting response... 200 OK
    Length: 3775 (3.7K) [text/plain]
    Saving to: ‘p0_data.txt.1’

    p0_data.txt.1       100%[===================>]   3.69K  --.-KB/s    in 0s

    2023-01-26 22:54:23 (52.3 MB/s) - ‘p0_data.txt.1’ saved [3775/3775]
```

```
#load the dataset
data = np.loadtxt('p0_data.txt', delimiter=',')

train_X = data[:, 0:2]
train_y = data[:, 2]

# Get the number of training examples and the number of features
m_samples, n_features = train_X.shape
print ("# of training examples = ", m_samples)
print ("# of features = ", n_features)

pos = np.where(train_y == 1)
neg = np.where(train_y == 0)
plt.scatter(train_X[pos, 0], train_X[pos, 1], marker='o', c='b')
plt.scatter(train_X[neg, 0], train_X[neg, 1], marker='x', c='r')
plt.xlabel('Exam 1 score')
plt.ylabel('Exam 2 score')
plt.legend(['Admitted', 'Not Admitted'])
plt.show()
```

```
# of training examples =  100
# of features =  2
```



3. Cost Function [5 points]

You're going to first implement the sigmoid function, then the cost function for (binary) logistic regression.

The sigmoid function is defined as $sigmoid(\mathbf{z}) = \frac{1}{1+e^{-\mathbf{z}}}$. It is important to handle potential underflow or overflow in the sigmoid implementation. It is also important to not take a log of 0.

Note that, you are asked to use the Numpy package for vector and matrix operations in order to ensure the **efficiency of the code**.

```
def sigmoid(z):
    """ Sigmoid function """
    ###################################################################
    # Compute the sigmoid function for the input here.                #
    ###################################################################

    s = None
    ### YOUR CODE HERE: be careful of the potential underflow or overflow here
    s = 1/(1+np.exp(-1*z))


    ### END YOUR CODE

    return s

# Check your sigmoid implementation
z = np.array([[1, 2], [-1, -2]])
f = sigmoid(z)
print (f)
```

```
    [[0.73105858 0.88079708]
     [0.26894142 0.11920292]]
```

```
def cost_function(theta, X, y):
    """ The cost function for logistic regression """
    ###############################################################################
    # Compute the cost given the current parameter theta on the training data set (
    ###############################################################################

    cost = None
    ### YOUR CODE HERE
    l = np.sum(theta*X, axis=1)
    y_pre = sigmoid(l)
    cost = np.sum(-y*np.log(y_pre+1e-8)-(1-y)*np.log(1-y_pre+1e-8))/y.shape[0]

    ### END YOUR CODE

    return cost

# Check your cost function implementation

t_X = np.array([[1, 2], [-1, -2]])
t_y = np.array([0, 1])
t_theta1 = np.array([-10, 10])
t_theta2 = np.array([10, -10])
t_c1 = cost_function(t_theta1, t_X, t_y)
t_c2 = cost_function(t_theta2, t_X, t_y)
print (t_c1)
print (t_c2)
```

```
9.999825148498655
4.5388898762821084e-05
```

4. Gradient Computation [5 points]

Implement the gradient computations for logistic regression.

```
def gradient_update(theta, X, y):
    """ The gradient update for logistic regression"""
    ###############################
    # Compute the gradient update #
    ###############################

    grad = None
    ### YOUR CODE HERE
    grad = []
    l = np.sum(theta*X, axis=1)
    y_pre = sigmoid(l) #h(x)
    for t in range(theta.shape[0]):
      g = (y_pre-y) @ X[:, t].T/y.shape[0]
      grad.append(g)
    grad = np.array(grad)
    ### END YOUR CODE

    return grad

# Check your gradient computation implementation
t_X = np.array([[1, 2, 3], [-1, -2, -3]])
t_y = np.array([0, 1])
t_theta1 = np.array([-10, 10, 0])
t_theta2 = np.array([10, -10, 0])
t_g1 = gradient_update(t_theta1, t_X, t_y)
t_g2 = gradient_update(t_theta2, t_X, t_y)
print (t_g1)
print (t_g2)
```

```
[0.9999546  1.9999092  2.99986381]
[4.53978687e-05 9.07957374e-05 1.36193606e-04]
```

5. Gradient Checking [Code provided. Bonus 5 points if implemented from scratch]

You can use the code provided below to check the gradient of your logistic regression functions using Scipy package. Alternatively, you can implement the gradient checking from scratch by yourself (bonus 5 points). If you attempt the bonus, your implementation should replicate the behavior Scipy's implementation. Note: Copying Scipy's implementation does not count.

Gradient checking is an important technique for debugging the gradient computation. Logistic regression is a relatively simple algorithm where it is straightforward to derive and implement its cost function and gradient computation. For more complex models, the gradient computation can be notoriously difficulty to debug and get right. Sometimes a subtly buggy implementation will manage to learn something that can look surprisingly reasonable, while performing less well than a correct implementation. Thus, even with a buggy implementation, it may not at all be apparent that anything is amiss.

```
def check_gradient(theta, X, y):
  eps = 1e-8
  # g = (cost_function(theta+eps, X, y) - cost_function(theta-eps, X, y))/(2*eps)
  g = []
  gradient = gradient_update(theta, X, y)
  for t in range(theta.shape[0]):
    t1 = theta.copy()
    t2 = theta.copy()
    t1[t] += eps
    t2[t] -= eps
    gi = (cost_function(t1, X, y)- cost_function(t2, X, y))/(2*eps)
    g.append(gi)

  dif = np.linalg.norm(gradient - np.array(g))

  return dif
```

```
# Check your gradient computation implementation
t_samples, t_features = 100, 10
t_X = np.random.randn(t_samples, t_features)
t_y = np.random.randint(2, size=t_samples)
t_theta = np.random.randn(t_features)

print("My check of gradient:", check_gradient(t_theta, t_X, t_y))

from scipy import optimize
print('Output of check_grad: %s' % optimize.check_grad(cost_function, gradient_upda
```

```
    My check of gradient: 3.3484683848134374e-06
    Output of check_grad: 3.379519151690403e-06
```

6. Gradient Descent and Decision Boundary [10 points]

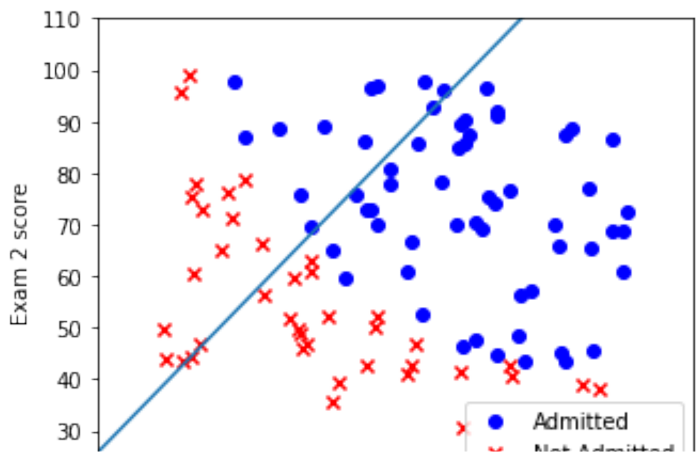
Implement the batch gradient descent algorithm for logistic regression. For every 'print_iterations' number of iterations, also visualize the decision boundary and observe how it changes during the training. Please print the change between **10-20** times to fully demonstrate the learned decision boundary, along with the allowing enough iterations for the method to converge.

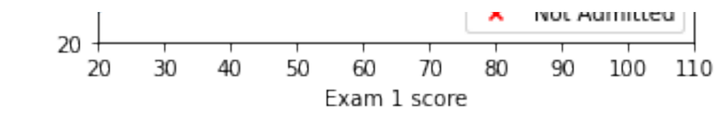
Please use the *x_axis_range* variable initialized at the end of the plotting code (on line 55 of the block below if line numbers are enabled) for the range of the x-axis when plotting.

Note that, you will need to carefully choose the learning rate and the total number of iterations (hint: without feature scaling, it will needs a small learning rate and a large number of iterations), especially given that the starter code does not include feature scaling (e.g., scale each feature by its maximum absolute value to convert feature value to [-1,1] range -- in order to make this homework simple and easier for you to write code to visualize.

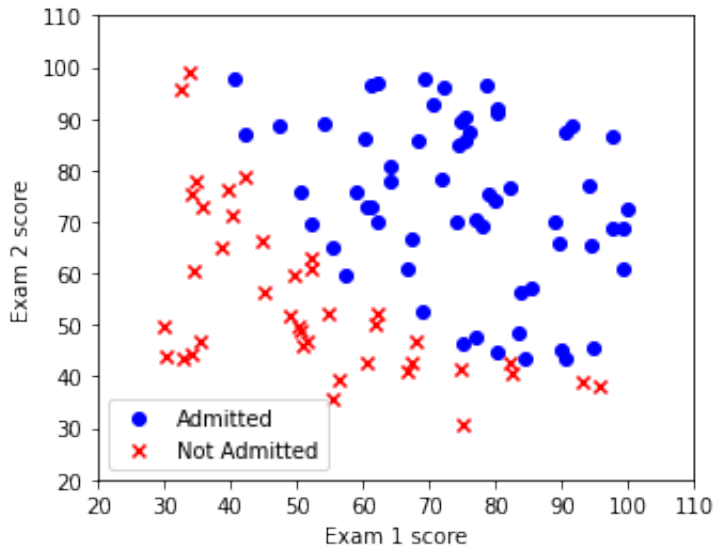
```
def gradient_descent(theta, X, y, alpha, max_iterations, print_iterations):
    """ Batch gradient descent algorithm """
    #################################################################
    # Update the parameter 'theta' iteratively to minimize the cost #
    # 'alpha' is learning rate.                                     #
    # Also visualize the decision boundary during learning          #
    #################################################################

    #alpha *= m_samples
    iteration = 0

    ### YOUR CODE HERE: handle bias term, i.e., adding x0=1 into the X
    # A side note: if doing feature scaling (not required in this homework), this w
    B = np.ones((X.shape[0], 1))
    x = np.concatenate((B, X), axis=1)

    ### END YOUR CODE



    while(iteration < max_iterations):
        iteration += 1

        ### YOUR CODE HERE: simultaneous update of partial gradients
        theta = theta - alpha * gradient_update(theta, x, y)




        ### END YOUR CODE


        # For first iteration and every print_iterations
        if iteration % print_iterations == 0 or iteration == 1:
            cost = 0

            ### YOUR CODE HERE: calculate the cost
            ### IMPORTANT: The cost function is guaranteed to decrease after
            ## every iteration of the gradient descent algorithm.
            cost = cost_function(theta, x, y)


            ### END YOUR CODE

            print ("[ Iteration", iteration, "]", "cost =", cost)
            plt.rcParams['figure.figsize'] = (5, 4)
            plt.xlim([20,110])
            plt.ylim([20,110])

            pos = np.where(y == 1)
            neg = np.where(y == 0)

            plt.scatter(X[pos, 0], X[pos, 1], marker='o', c='b')
            plt.scatter(X[neg, 0], X[neg, 1], marker='x', c='r')
            plt.xlabel('Exam 1 score')
            plt.ylabel('Exam 2 score')
            plt.legend(['Admitted', 'Not Admitted'])
            x_axis_range = np.arange(10, 100, 0.1)

            ### YOUR CODE HERE: plot the decision boundary using the variable 'x_ax

            plot_y = np.array(-(theta[0] +theta[1] * x_axis_range)/theta[2])
            plt.plot(x_axis_range, plot_y, label = "Decision_Boundary")


            ### END YOUR CODE

            plt.show()

    return theta


### YOUR CODE HERE: initialize the parameters 'theta' to random values;
### And set up learning rate, number of max iterations, number of iterations for pr


initial_theta = np.random.randn(3)
alpha_test = 0.001
max_iter = 100000
print_iter = 10000


### END YOUR CODE


learned_theta = gradient_descent(initial_theta, train_X, train_y, alpha_test, max_i
```

```
    [ Iteration 1 ] cost = 10.2031499837866
```

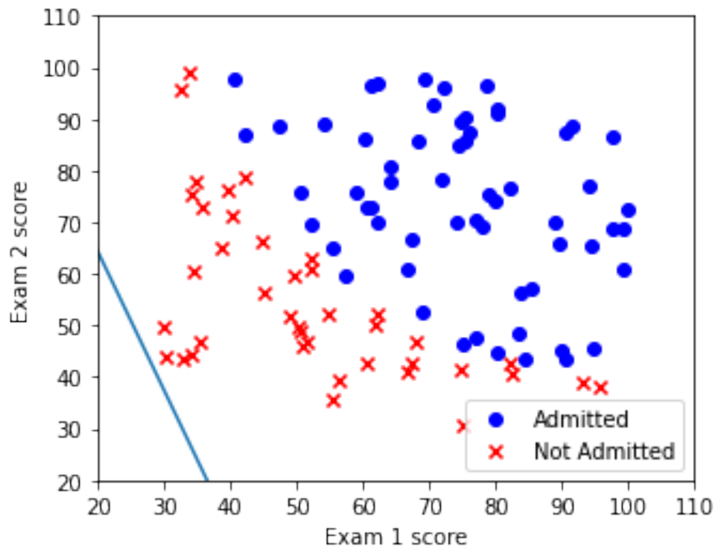




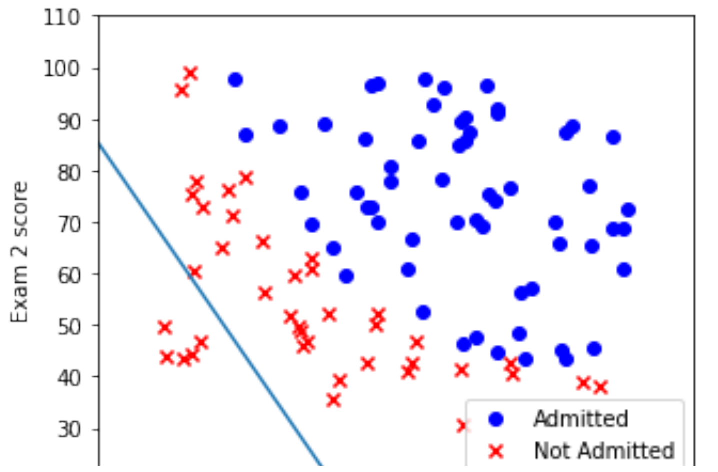
```
[ Iteration 10000 ] cost = 0.6304760059262939
```

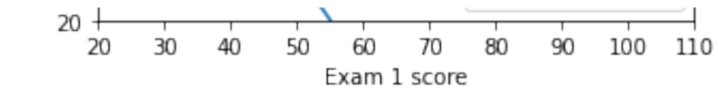


```
[ Iteration 20000 ] cost = 0.5855791102754847
```

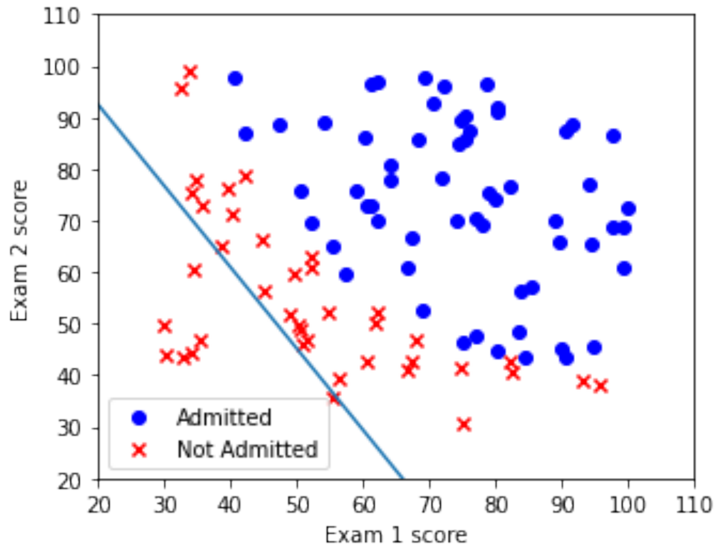


```
[ Iteration 30000 ] cost = 0.5477607398143982
```

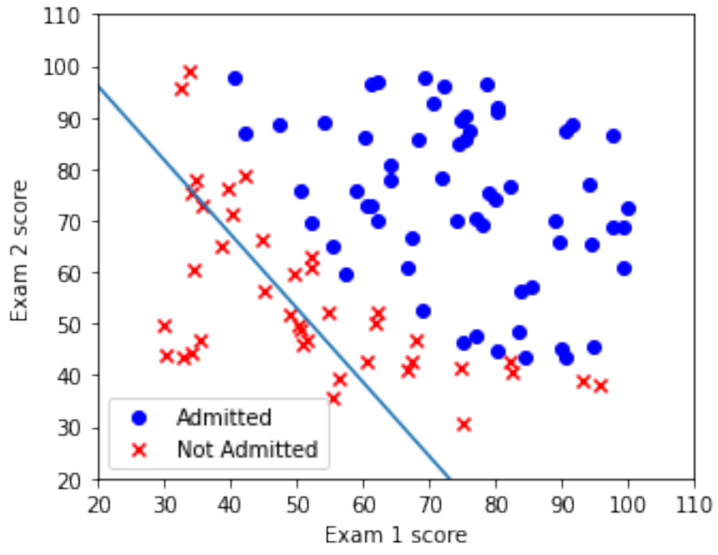




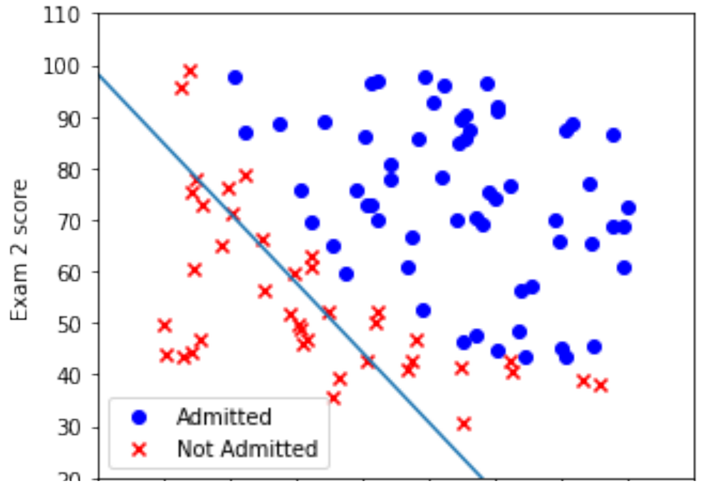
```
[ Iteration 40000 ] cost = 0.5157960900726924
```

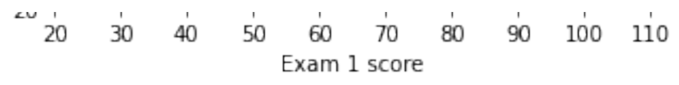


```
[ Iteration 50000 ] cost = 0.4886318753797903
```

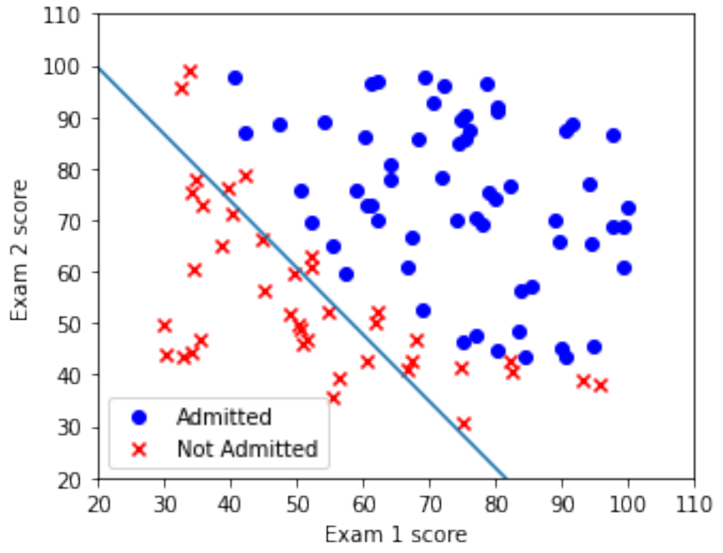


```
[ Iteration 60000 ] cost = 0.46539322001973404
```

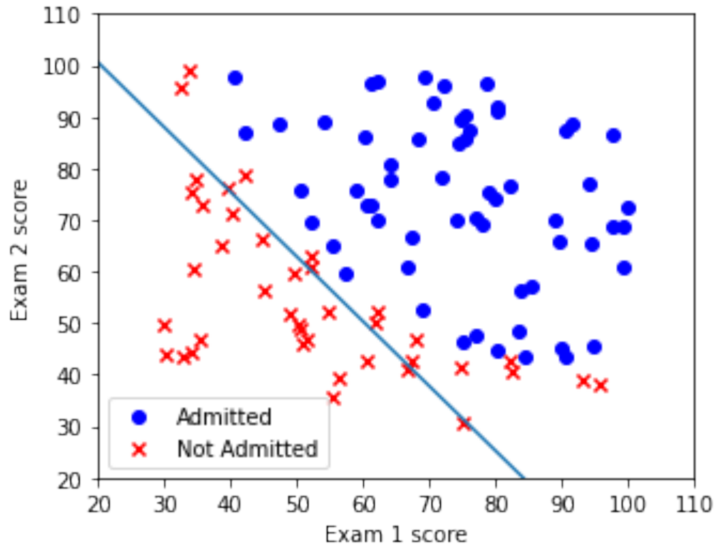




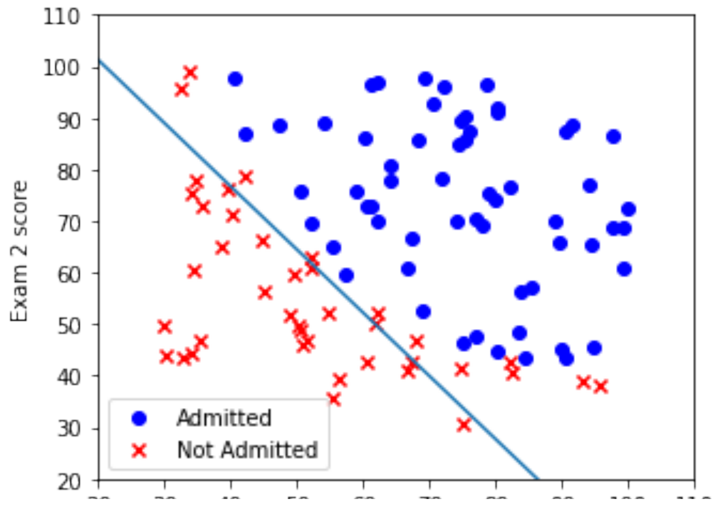
```
[ Iteration 70000 ] cost = 0.4453684933857766
```

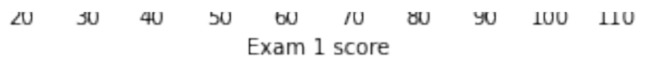


```
[ Iteration 80000 ] cost = 0.42798530562250797
```

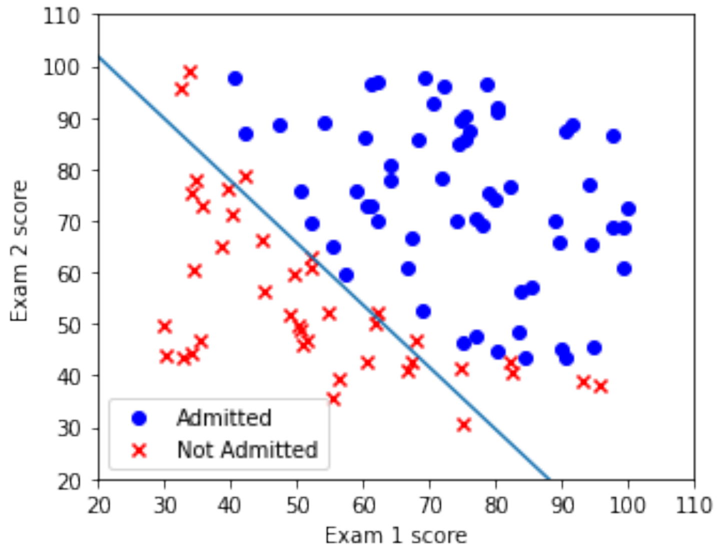


```
[ Iteration 90000 ] cost = 0.41278534201290407
```





```
[ Iteration 100000 ] cost = 0.399401641037959
```



6. Predicting [5 points]

Now that you learned the parameters of the model, you can use the model to predict whether a particular student will be admited.

```
def predict(theta, X):
    """ Predict label using learned logistic regression parameters """
    ##################################################################################
    # Predict a label of 0 or 1 using learned logistic regression parameters for 
    # Return the probabilities of X as a (1, N) array of floats
    # Return the predicted_labels as a (1, N) array of 0 or 1 integers
    ##################################################################################

    ### YOUR CODE HERE:
    B = np.ones((X.shape[0], 1))
    x = np.concatenate((B, X), axis=1)

    l = np.sum(theta*x, axis=1)
    probabilities = sigmoid(l)

    predicted_labels = np.round(probabilities)

    ### END YOUR CODE


    ## convert an array of booleans 'predicted_labels' into an array of 0 or 1 in
    return probabilities, 1*predicted_labels

# Check your predication function implementation
t_X1 = np.array([[90, 90]])
t_X2 = np.array([[50, 60]])
t_X3 = np.array([[10, 50]])
print (predict(learned_theta, t_X1))
print (predict(learned_theta, t_X2))
print (predict(learned_theta, t_X3))

# Compute accuracy on the training dateset
t_prob, t_label = predict(learned_theta, train_X)
t_precision = t_label[np.where(t_label == train_y)].size / float(train_y.size) *
print('Accuracy on the training set: %s%%' % round(t_precision,2))
```

```
(array([0.92865918]), array([1.]))
(array([0.4493341]), array([0.]))
(array([0.09414432]), array([0.]))
Accuracy on the training set: 90.0%
```

7. Submit Your Homework

This is the end. Congratulations!

Now, follow the steps below to submit your homework in Gradescope:

1. rename this ipynb file to 'CS4650_p0_GTusername.ipynb'. Make sure all cells have been run.
2. click on the menu 'File' --> 'Download' --> 'Download .py'.
3. click on the menu 'File' --> 'Download' --> 'Download .ipynb'.
4. Download the notebook as a .pdf document. Ensure the output from step 5. Gradient Descent and Decision Boundary is captured. This question cannot be graded if the output from this cell is not captured.
5. Upload all 3 files to GradeScope. Double check the files start with 'CS4650_p0_*', capitalization matters.

NOTE: If you experience issues when uploading to GradeScope and did not use Google Colab, we ask that you upload your notebook to Colab, re-run all cells, download the notebook and Python script, and re-upload them to GradeScope as students have reported issues in the past when using local Juypter Notebook instances.

✓ 0s completed at 8:37 PM