1. Linear Discriminant Analysis (LDA) [50 pts]

In this part of the exercise, you will re-visit the problem of predicting whether a student gets admitted into a university. However, in this part, you will build a linear discriminant analysis (LDA) classifier for this problem.

LDA is a generative model for classification that assumes the class covariances are equal. Given a training dataset of positive and negative features (x, y) with y ∈ {0, 1} , LDA models the data x as generated from class-conditional Gaussians:

$P(x, y) = P(x|y)P(y)$ where $P(y = 1) = \pi$ and $P(x|y) = N(x; \mu^y, \Sigma)$

where means μ^y are class-dependent but the covariance matrix Σ is class-independent (the same for all classes).

A novel feature x is classified as a positive if $P(y = 1|x) > P(y = 0|x)$, whichis equivalent to $a(x) > 0$, where the linear classifier $a(x) = w^T x + w_0$ has weights given by $w = \Sigma^{-1}(\mu^1 - \mu^0)$.

In practice, and in this assignment, we use $a(x) >$ some threshold, or equivalently, $w^T x > T$ for some constant T.

As we saw in lecture, LDA and logistic regression can be expressed in the same form

$P(y = 1|x) = \frac{1}{1+e^{-\theta^T x}}$.

However, they generally produce different solutions for the parameter theta.

Implementation

In this assignment, you can assume the prior probabilities for the two classes are the same (although the number of the positive and negative samples in the training data is not the same), and that the threshold T is zero. As a bonus, you are encouraged to explore how the different prior probabilities shift the decision boundary.

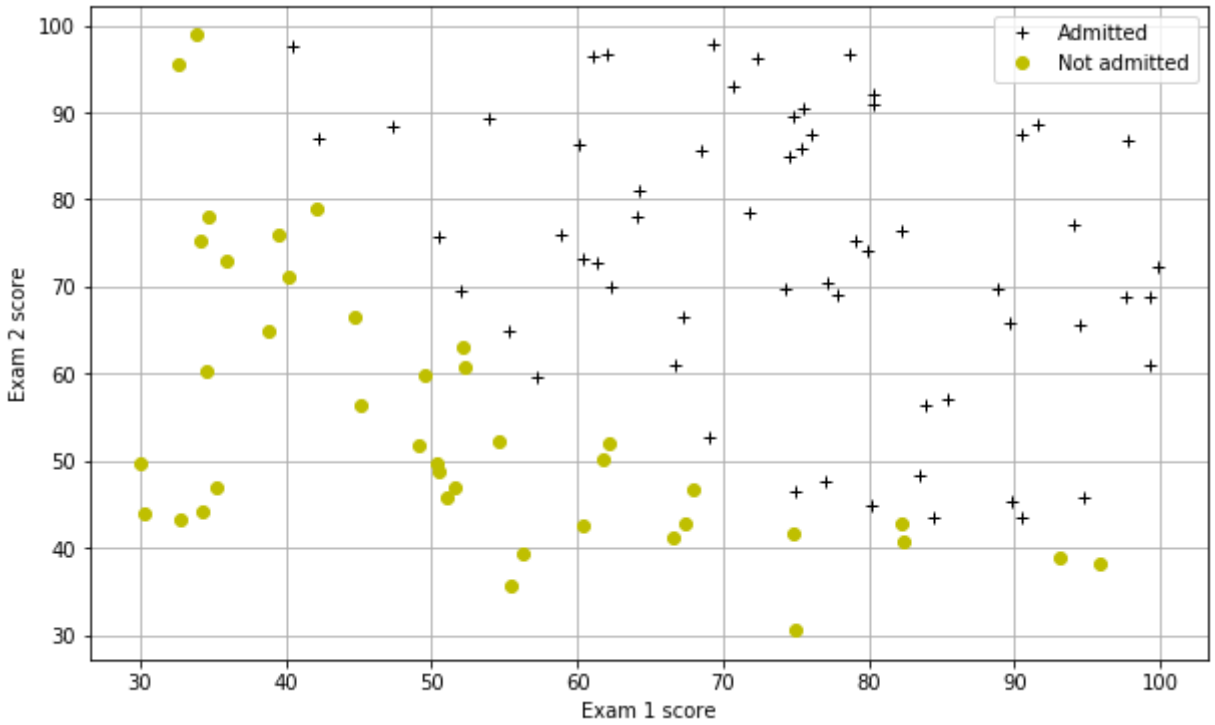
In [7]:
```
atplotlib inline
port numpy as np
port matplotlib.pyplot as plt
port pandas as pd

tafile = 'data/ex2data1.txt'
head $datafile
ls = np.loadtxt(datafile,delimiter=',',usecols=(0,1,2),unpack=True) #Read ir
Form the usual "X" matrix and "y" vector
= np.transpose(np.array(cols[:-1]))
= np.transpose(np.array(cols[-1:]))
= y.size # number of training examples
Insert the usual column of 1's into the "X" matrix
= np.insert(X,0,1,axis=1)

ivide the sample into two: ones with positive classification, one with null
s = np.array([X[i] for i in range(X.shape[0]) if y[i] == 1])
g = np.array([X[i] for i in range(X.shape[0]) if y[i] == 0])

f plotData():
  plt.figure(figsize=(10,6))
  plt.plot(pos[:,1],pos[:,2],'k+',label='Admitted')
  plt.plot(neg[:,1],neg[:,2],'yo',label='Not admitted')
  plt.xlabel('Exam 1 score')
  plt.ylabel('Exam 2 score')
  plt.legend()
  plt.grid(True)

otData()
```



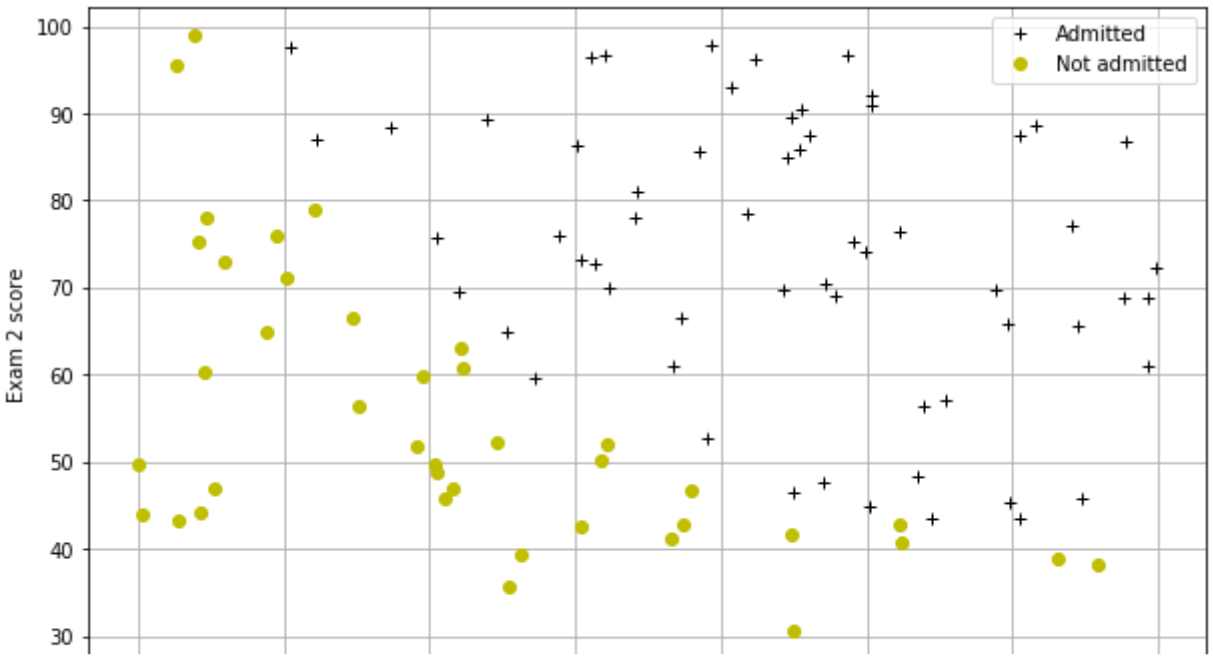
Implement the LDA classifier by completing the code here. As an implementation detail, you should first center the positive and negative data separately, so that each has a mean equal to 0, before computing the covariance, as this tends to give a more accurate estimate.

You should center the whole training data set before applying the classifier. Namely, subtract the middle value of the two classes' means ($\frac{1}{2}$(pos mean+neg mean)), which is on the separating plane when their prior probabilities are the same and becomes the 'center' of the data. [5 pts]

In [12]:
```
# IMPLEMENT THIS
pos_mean = np.mean(pos,axis=0)
neg_mean = np.mean(neg,axis=0)

pos_data = pos - 0.5*(pos_mean+neg_mean)
neg_data = neg - 0.5*(pos_mean+neg_mean)

plotData()
```



Implement the LDA algorithm here (Compute the covariance on all data): [10 pts each for getting cov_all, w and y_lda]

In [21]:
```
# IMPLEMENT THIS
X_data = np.concatenate((neg_data,pos_data),axis=0)

cov_all = np.cov(pos_data.T[1:]) + np.cov(neg_data.T[1:]) # SHAPE: (2,2)
print('cov_all shape:',cov_all.shape)
print('cov_all:',cov_all)

w = np.linalg.inv(cov_all)@((pos_mean-neg_mean)[1:]) # w=cov_all^(-1)(pos_m
print('w:',w)

y_lda =  np.dot(X_data[:,1:],w) # SHAPE: (100,)
print('y_lda shape:',y_lda.shape)
print('y_lda:',y_lda)
```

```
cov_all shape: (2, 2)
cov_all: [[ 530.34940218 -250.35734987]
 [-250.35734987  515.01463225]]
w: [0.0785182  0.07571361]
y_lda shape: (100,)
y_lda: [-1.217527   -4.14213898 -1.50928301 -2.03987204  0.57740491 -1.63
662473
 -0.98214493 -0.97437751 -1.31026252 -2.17669519 -3.81011326  0.40526738
 -1.18377781 -1.87719649 -0.97260056 -1.29595002 -1.60049371  0.3051004
 -0.82365479 -1.4630883  -2.24855258 -0.30084464 -2.36606595 -1.0163075
 -2.56001785 -2.12435908 -1.4218374  -0.04763126 -3.51295078 -2.45406938
 -3.72861798 -1.3051982  -1.50617296 -2.06229106 -3.99536535 -1.87831908
 -0.15287996 -0.57033324 -1.13229596 -2.79409991  1.41651591  2.06645738
  2.26156828 -0.42806524  2.75042118  0.08193777  2.4096817   3.00137212
  1.14827486 -0.42721089  2.74041813 -0.19726214  2.17952099  1.49696021
  0.34226688 -0.15364966  0.48829374  1.18177521 -0.5001216   1.32125746
  1.00750479  2.53633745  1.55205728  4.39968104  2.35693677  4.06138802
  2.04863516  2.56951142  0.55117848  3.03707215  1.27289839  1.73368585
  2.5694766   0.71774956  3.43420168  0.01506522  1.09266742  3.12260948
```

Completing the code to compute the training set accuracy. You should get a training accuracy around 89%. [5 pts]

In [22]:
```
# IMPLEMENT THIS
y_lda[y_lda<0]=0
y_lda[y_lda>0]=1

my_label = np.sort(y,axis=None)

counter = 0
for i in range(m):
    if my_label[i] == y_lda[i]:
        counter+=1

accuracy = counter/m
print(accuracy)
```

```
0.89
```

Written Problem [10 pts]

Yunmo Ni

Writer Problem

$P(C_1) = P(C_2) = \frac{1}{2}$

$$a = \log \frac{P(x|C_1)P(C_1)}{P(x|C_2)P(C_2)}$$

$$= \log P(x|C_1) - \log P(x|C_2) + \log \frac{P(C_1)}{P(C_2)}$$

$$= \log P(x|C_1) - \log P(x|C_2)$$

$$= -\log (2\pi)^{\frac{p}{2}} |\Sigma_1|^{\frac{1}{2}} - \frac{1}{2}(x-\mu_1)^T \Sigma_1^{-1}(x-\mu_1) + \log (2\pi)^{\frac{p}{2}} |\Sigma_2|^{\frac{1}{2}} + \frac{1}{2}(x-\mu_2)^T \Sigma_2^{-1}(x-\mu_2)$$

$$= -\frac{1}{2}(x-\mu_1)^T \Sigma^{-1}(x-\mu_1) + \frac{1}{2}(x-\mu_2)^T \Sigma^{-1}(x-\mu_2)$$

$$= \frac{1}{2}(x^T \Sigma^{-1} x - 2x^T \Sigma^{-1} \mu_1 + \mu_1^T \Sigma^{-1} \mu_1) + \frac{1}{2}(x^T \Sigma^{-1} x - 2x^T \Sigma^{-1} \mu_2 + \mu_2^T \Sigma^{-1} \mu_2)$$

$$= -\frac{1}{2}(-2x^T \Sigma^{-1} \mu_1 + 2x^T \Sigma^{-1} \mu_2 + \mu_1^T \Sigma^{-1} \mu_1 - \mu_2^T \Sigma^{-1} \mu_2)$$

$$= -\frac{1}{2}(-2x^T \Sigma^{-1}(\mu_1 - \mu_2) + \mu_1^T \Sigma^{-1} \mu_1 - \mu_2^T \Sigma^{-1} \mu_2)$$

$$= x^T \Sigma^{-1}(\mu_1 - \mu_2) + \underline{\theta_0} \text{ (constant)}$$

Suppose $\theta = \Sigma^{-1}(\mu_1 - \mu_2)$

$\therefore a(x) = \theta^T x + \theta_0$ linear

ANSWER:

2. CNN on MNIST using TensorFlow™ [50 pts]

Note 1: The following has been verified to work with TensorFlow version 1.11-1.14*. As before, you ca use any of the following options to run the notebook with:

- Your own installation of TensorFlow 1.11-1.14 using e.g. Conda
- Colab - has a 12-hour limit for running the compute instance. GPU instances are also available:
  `Runtime -> Change runtime type -> Hardware accelerator`
- Shared Computing Cluster with GPUs: port forwarding or easier "SCC OnDemand" option via
  `scc5.bu.edu`

* *Adapted from official TensorFlow™ tour guide.*

TensorFlow is a powerful library for doing large-scale numerical computation. One of the tasks at whi excels is implementing and training deep neural networks. In this assignment you will learn the basic building blocks of a TensorFlow model while constructing a deep convolutional MNIST classifier.

What you are expected to implement in this tutorial:

- Create a softmax regression function that is a model for recognizing MNIST digits, based on loo at every pixel in the image
- Use Tensorflow to train the model to recognize digits by having it "look" at thousands of exampl
- Check the model's accuracy with MNIST test data
- Build, train, and test a multilayer convolutional neural network to improve the results

Here is a diagram, created with TensorBoard, of the model we will build:

 tensorflow graph

Implement Utilities

Weight Initialization

To create this model, we're going to need to create a lot of weights and biases. One should generally initialize weights with a small amount of noise for symmetry breaking, and to prevent 0 gradients. Sin we're using ReLU neurons, it is also good practice to initialize them with a slightly positive initial bias t avoid "dead neurons". Instead of doing this repeatedly while we build the model, let's create two handy functions to do it for us.

```
import tempfile

import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

tf.logging.set_verbosity(tf.logging.ERROR)

def weight_variable(shape):
    """weight_variable generates a weight variable of a given shape."""
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)


def bias_variable(shape):
    """bias_variable generates a bias variable of a given shape."""
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

The default version of TensorFlow in Colab will soon switch to TensorFlow 2.x.
We recommend you upgrade now or ensure your notebook will continue to use TensorFlow 1.x via the `%tensorflow_version 1.x` magic: more info.

Convolution and Pooling [5 pts]

Our convolutions uses a stride of one and are zero padded so that the output is the same size as the input. Our pooling is plain old max pooling over 2x2 blocks.

NOTE: FOR ALL THE FOLLOWING CODES, DO NOT IMPLEMENT YOUR OWN VERSION. USE THE BUILT METHODS FROM TENSORFLOW.

Take a look at TensorFlow API Docs.

```
# IMPLEMENT THIS
def conv2d(x, W):
    conv2d = tf.layers.conv2d(inputs=x,filters=W, kernel_size=[5,5], padding="same",)
    return conv2d


def max_pool_2x2(x):
      max_pool = tf.layers.max_pooling2d(inputs=x, pool_size=[2,2],strides=2)
      return max_pool
```

Build the CNN

First Convolutional Layer[10 pts]

We can now implement our first layer. It will consist of convolution, followed by max pooling. The convolution will compute 32 features for each 5x5 patch. Its weight tensor will have a shape of [5, 5, 1

32]. The first two dimensions are the patch size, the next is the number of input channels, and the last the number of output channels. We will also have a bias vector with a component for each output cha

To apply the layer, we first reshape x to a 4d tensor, with the second and third dimensions correspondi to image width and height, and the final dimension corresponding to the number of color channels.

We then convolve x_image with the weight tensor, add the bias, apply the ReLU function, and finally m pool. The max_pool_2x2 method will reduce the image size to 14x14.

Second Convolutional Layer[5 pts]

In order to build a deep network, we stack several layers of this type. The second layer will have 64 features for each 5x5 patch.

Fully Connected Layer[10 pts]

Now that the image size has been reduced to 7x7, we add a fully-connected layer with 1024 neurons t allow processing on the entire image. We reshape the tensor from the pooling layer into a batch of vectors, multiply by a weight matrix, add a bias, and apply a ReLU.

SoftmaxLayer[5 pts]

Finally, we add a layer of softmax regression.

```
def deepnn(x):
    """
    deepnn builds the graph for a deep net for classifying digits.
    Args:
      x: an input tensor with the dimensions (N_examples, 784), where 784 is the
      number of pixels in a standard MNIST image.
    Returns:
      A tuple (y, keep_prob). y is a tensor of shape (N_examples, 10), with values
      equal to the logits of classifying the digit into one of 10 classes (the
      digits 0-9). keep_prob is a scalar placeholder for the probability of
      dropout.
    """
    # Reshape to use within a convolutional neural net.
    # Last dimension is for "features" - there is only one here, since images are
    # grayscale -- it would be 3 for an RGB image, 4 for RGBA, etc.
    with tf.name_scope('reshape'):
        x_image = tf.reshape(x,[-1,28,28,1])#None

    # First convolutional layer - maps one grayscale image to 32 feature maps.
    with tf.name_scope('conv1'):
        h_conv1 = tf.nn.relu(conv2d(x_image,32))

    # Pooling layer - downsamples by 2X.
    with tf.name_scope('pool1'):
        h_pool1 = max_pool_2x2(h_conv1)
```

```
    # Second convolutional layer -- maps 32 feature maps to 64.
    with tf.name_scope('conv2'):
        h_conv2 = tf.nn.relu(conv2d(h_pool1,64))

    # Second pooling layer.
    with tf.name_scope('pool2'):
        h_pool2 = max_pool_2x2(h_conv2)

    # Fully connected layer 1 -- after 2 round of downsampling, our 28x28 image
    # is down to 7x7x64 feature maps -- maps this to 1024 features.
    with tf.name_scope('fc1'):
        h_pool2_flat = tf.contrib.layers.flatten(h_pool2,scope='pool2flat')
        h_fc1 = tf.layers.dense(inputs=h_pool2_flat,units=1024,activation=tf.nn.relu)

    # Map the 1024 features to 10 classes, one for each digit
    with tf.name_scope('fc2'):
        y_conv = tf.layers.dense(inputs=h_fc1,units=10)
    return y_conv
```

Complete the Graph[10 pts]

We start building the computation graph by creating nodes for the input images and target output cla

```
# Import data
mnist = input_data.read_data_sets('/tmp/tensorflow/mnist/input_data', one_hot=True)

x = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.float32, [None, 10])

# Build the graph for the deep net
y_conv = deepnn(x)
```

```
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting /tmp/tensorflow/mnist/input_data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/tensorflow/mnist/input_data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/tensorflow/mnist/input_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/tensorflow/mnist/input_data/t10k-labels-idx1-ubyte.gz
```

We can specify a loss function just as easily. Loss indicates how bad the model's prediction was on a single example; we try to minimize that while training across all the examples. Here, our loss function the cross-entropy between the target and the softmax activation function applied to the model's prediction. As in the beginners tutorial, we use the stable formulation:

```
with tf.name_scope('loss'):
    my_ydict = dict(labels=y_,logits=y_conv)
    my_loss = tf.nn.softmax_cross_entropy_with_logits(**my_ydict)
```

```
    cross_entropy = tf.reduce_mean(my_loss)

with tf.name_scope('adam_optimizer'):
    my_trainer = tf.train.AdamOptimizer(learning_rate=0.001)
    train_step = my_trainer.minimize(cross_entropy)
```

First we'll figure out where we predicted the correct label. tf.argmax is an extremely useful function wh gives you the index of the highest entry in a tensor along some axis. For example, tf.argmax(y,1) is the label our model thinks is most likely for each input, while tf.argmax(y_,1) is the true label. We can use tf.equal to check if our prediction matches the truth.

That gives us a list of booleans. To determine what fraction are correct, we cast to floating point numb and then take the mean. For example, [True, False, True, True] would become [1,0,1,1] which would become 0.75.

```
with tf.name_scope('accuracy'):
     my_ypred = tf.argmax(tf.nn.softmax(y_conv),axis=1)
     my_yreal = tf.argmax(y_,axis=1)
     correct_prediction = tf.equal(tf.cast(my_ypred,tf.int64),my_yreal)
     accuracy = tf.reduce_mean(tf.cast(correct_prediction,tf.float32))


# For saving the graph, DO NOT CHANGE.
graph_location = tempfile.mkdtemp()
print('Saving graph to: %s' % graph_location)
train_writer = tf.summary.FileWriter(graph_location)
train_writer.add_graph(tf.get_default_graph())
```

```
Saving graph to: /tmp/tmpub08g5pu
```

Train and Evaluate the Model[5 pts]

We will use a more sophisticated ADAM optimizer instead of a Gradient Descent Optimizer.

We will add logging to every 100th iteration in the training process.

Feel free to run this code. Be aware that it does 20,000 training iterations and may take a while (possibl up to half an hour), depending on your processor. With GPU acceleration it should be much faster.

The final test set accuracy after running this code should be approximately 99.2%.

We have learned how to quickly and easily build, train, and evaluate a fairly sophisticated deep learning model using TensorFlow.

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(2000):
        batch = mnist.train.next_batch(50)
        if i % 100 == 0:
```

```
        if i % 100 == 0:
            train_accuracy = accuracy.eval(feed_dict={x: batch[0], y_: batch[1]})
            print('step %d, training accuracy %g' % (i, train_accuracy))
        train_step.run(feed_dict={x: batch[0], y_: batch[1]})

    print('test accuracy %g' % accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels}
```

```
step 0, training accuracy 0.06
step 100, training accuracy 0.98
step 200, training accuracy 0.96
step 300, training accuracy 0.96
step 400, training accuracy 0.98
step 500, training accuracy 0.98
step 600, training accuracy 0.98
step 700, training accuracy 0.98
step 800, training accuracy 0.96
step 900, training accuracy 0.96
step 1000, training accuracy 1
step 1100, training accuracy 0.98
step 1200, training accuracy 1
step 1300, training accuracy 0.98
step 1400, training accuracy 1
step 1500, training accuracy 1
step 1600, training accuracy 0.98
step 1700, training accuracy 0.98
step 1800, training accuracy 1
step 1900, training accuracy 1
test accuracy 0.9889
```