Logistic regression derived from Bernoulli probabilistic model MLE

The hypothesis function for logistic regression is sigmoid function.

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$$

For such a function, we cannot simply define a square-error like cost function as in linear regression and then minimize it to find the solution. The reason is that such a cost function is not convex and thus will introduce complexity in solving the problem. Thus a probabilistic model Bernoulli distribution (not like the Gaussian in derivation of linear regression) will be used to derive the update rule by maximizing log likelihood.

$$p(y \mid x; \theta) = (h_\theta(x))^y (1 - h_\theta(x))^{1-y}$$

Further assuming there are m independently generated samples, then the likelihood can be written as

$$L(\theta) = \prod_{i=1}^{m} p(y^{(i)} \mid x^{(i)}; \theta) = \prod_{i=1}^{m} (h_\theta(x^{(i)}))^{y^{(i)}} (1 - h_\theta(x^{(i)}))^{1-y^{(i)}}$$

It is easier to maximize the log likelihood

$$l(\theta) = \log L(\theta) = \sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))$$

In a few cases we can obtain the closed form of parameter θ by maximizing the likelihood. We first calculate the gradient of likelihood w.r.t. the θ and set it to be zero, and then solve the equation to obtain the solution to the parameter. We can do this for example in estimating the parameters of mixture of Gaussian/Naive Bayes models with labels (supervised learning). However, in most cases, we must employ a gradient ascend (or descend for minimizing cost function) approach to iteratively obtain the correct parameter. For logistic regression, the gradient ascent rule obtained from maximizing the likelihood has the form

$$\theta_j := \theta_j + \alpha \frac{\partial l(\theta)}{\partial \theta_j} = \theta_j + \alpha (y^{(i)} - h_\theta(x^{(i)}) x_j^{(i)}$$

The derivation of a vectorized version for calculating logistic regression has been done in the notes for deep learning. This is for fast numerical calculation and for connecting traditional machine learning algorithm to neural network.

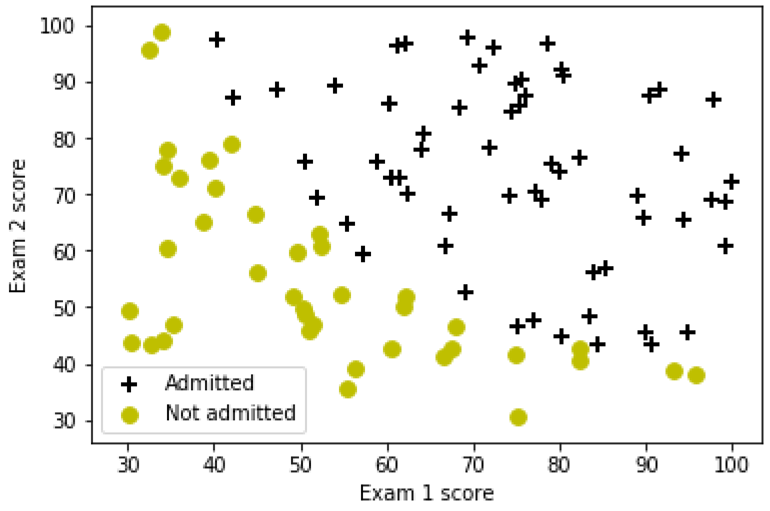
Exercise 1 --

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from sklearn.preprocessing import PolynomialFeatures

#from sklearn.preprocessing import PolynomialFeatures

# pd.set_option('display.notebook_repr_html', False)
# pd.set_option('display.max_columns', None)
# pd.set_option('display.max_rows', 150)
# pd.set_option('display.max_seq_items', None)

%matplotlib inline

# import seaborn as sns
# sns.set_context('notebook')
# sns.set_style('white')
```

In [3]:
```python
def loaddata(file, delimeter):
    data = np.loadtxt(file, delimiter=delimeter)
    print('Dimensions: ',data.shape)
    print(data[1:6,:])
    return(data)

def plotData(data, label_x, label_y, label_pos, label_neg, axes=None):
    # Get indexes for class 0 and class 1
    neg = data[:,2] == 0
    pos = data[:,2] == 1

    # If no specific axes object has been passed, get the current axes.
    if axes == None:
        axes = plt.gca()
    axes.scatter(data[pos][:,0], data[pos][:,1], marker='+', c='k', s=60, linewidth=2, label=label_pos)
    axes.scatter(data[neg][:,0], data[neg][:,1], c='y', s=60, label=label_neg)
    axes.set_xlabel(label_x)
    axes.set_ylabel(label_y)
    axes.legend(frameon= True, fancybox = True);

data = loaddata('ex2data1.txt', ',')
```

```
Dimensions:  (100, 3)
[[30.28671077 43.89499752  0.        ]
 [35.84740877 72.90219803  0.        ]
 [60.18259939 86.3085521   1.        ]
 [79.03273605 75.34437644  1.        ]
 [45.08327748 56.31637178  0.        ]]
```

In [4]:
```
X = np.c_[np.ones((data.shape[0],1)), data[:,0:2]]
y = np.c_[data[:,2]]

plotData(data, 'Exam 1 score', 'Exam 2 score', 'Admitted', 'Not admitted')
```



```
In [5]: def sigmoid(z):
            return(1 / (1 + np.exp(-z)))

        def costFunction(theta, X, y):
            m = y.size
            h = sigmoid(X.dot(theta))

            J = -1*(1/m)*(np.log(h).T.dot(y)+np.log(1-h).T.dot(1-y))

            if np.isnan(J[0]):
                return(np.inf)
            return(J[0])

        def gradient(theta, X, y):
            m = y.size
            h = sigmoid(X.dot(theta.reshape(-1,1)))

            grad =(1/m)*X.T.dot(h-y)

            return(grad.flatten())
```

```
In [6]: initial_theta = np.zeros(X.shape[1])
        cost = costFunction(initial_theta, X, y)
        grad = gradient(initial_theta, X, y)
        print('Cost: \n', cost)
        print('Grad: \n', grad)
```

```
Cost: 
 0.6931471805599452
Grad: 
 [ -0.1        -12.00921659 -11.26284221]
```

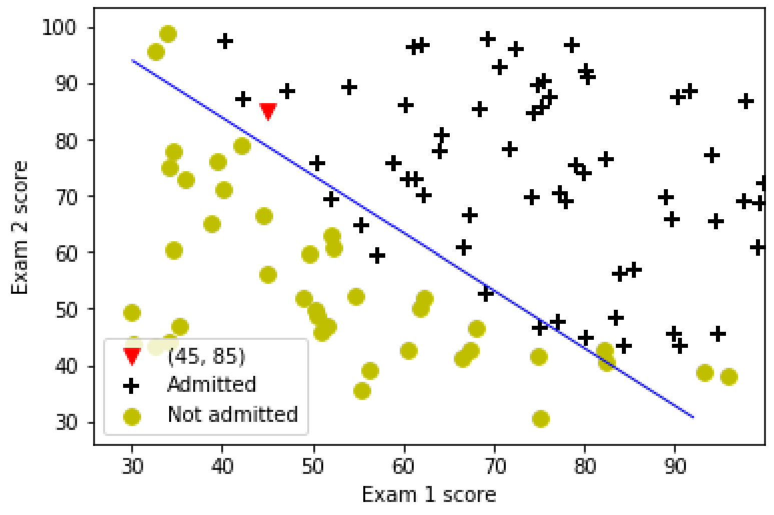
```
In [ ]: res = minimize(costFunction, initial_theta, args=(X,y), method=None, jac=gradient, options={'maxiter':400})
        res
```

In the calculation of cost function, we need add an extra small term to avoid the log (0) case.

In [8]:
```
def predict(theta, X, threshold=0.5):
    p = sigmoid(X.dot(theta.T)) >= threshold
    return(p.astype('int'))

# Student with Exam 1 score 45 and Exam 2 score 85
# Predict using the optimized Theta values from above (res.x)
sigmoid(np.array([1, 45, 85]).dot(res.x.T))

p = predict(res.x, X)
print('Train accuracy {}%'.format(100*sum(p == y.ravel())/p.size))
```

```
Train accuracy 89.0%
```

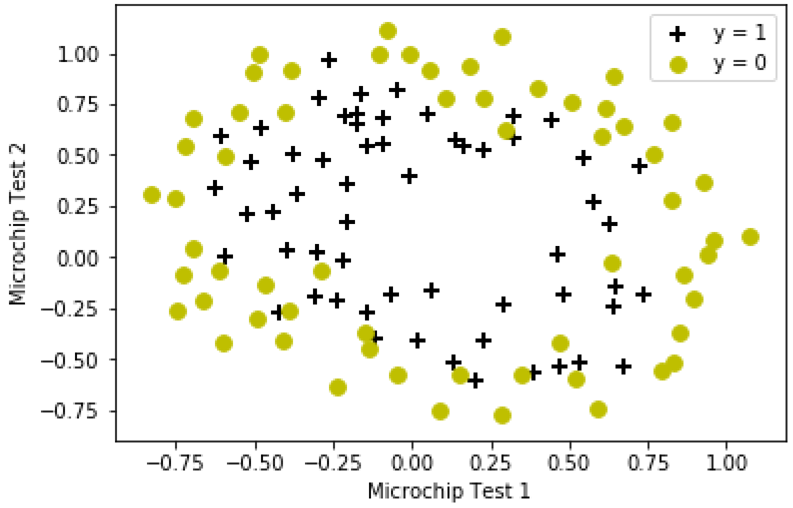
In [10]:
```
plt.scatter(45, 85, s=60, c='r', marker='v', label='(45, 85)')
plotData(data, 'Exam 1 score', 'Exam 2 score', 'Admitted', 'Not admitted')
x1_min, x1_max = X[:,1].min(), X[:,1].max(),
x2_min, x2_max = X[:,2].min(), X[:,2].max(),
xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max), np.linspace(x2_min, x2_max))
h = sigmoid(np.c_[np.ones((xx1.ravel().shape[0],1)), xx1.ravel(), xx2.ravel()].dot(res.x))
h = h.reshape(xx1.shape)
plt.contour(xx1, xx2, h, [0.5], linewidths=1, colors='b');
```



In [ ]:

In [11]:
```
#Regularized logistic regression
data2 = loaddata('ex2data2.txt', ',')

y = np.c_[data2[:,2]]
X = data2[:,0:2]

plotData(data2, 'Microchip Test 1', 'Microchip Test 2', 'y = 1', 'y = 0')
```

```
Dimensions:  (118, 3)
[[-0.092742  0.68494   1.        ]
 [-0.21371   0.69225   1.        ]
 [-0.375     0.50219   1.        ]
 [-0.51325   0.46564   1.        ]
 [-0.52477   0.2098    1.        ]]
```



In [12]:
```
# Note that this function inserts a column with 'ones' in the design matrix for the intercept.
poly = PolynomialFeatures(6)
XX = poly.fit_transform(data2[:,0:2])
XX.shape
```

Out[12]:
```
(118, 28)
```

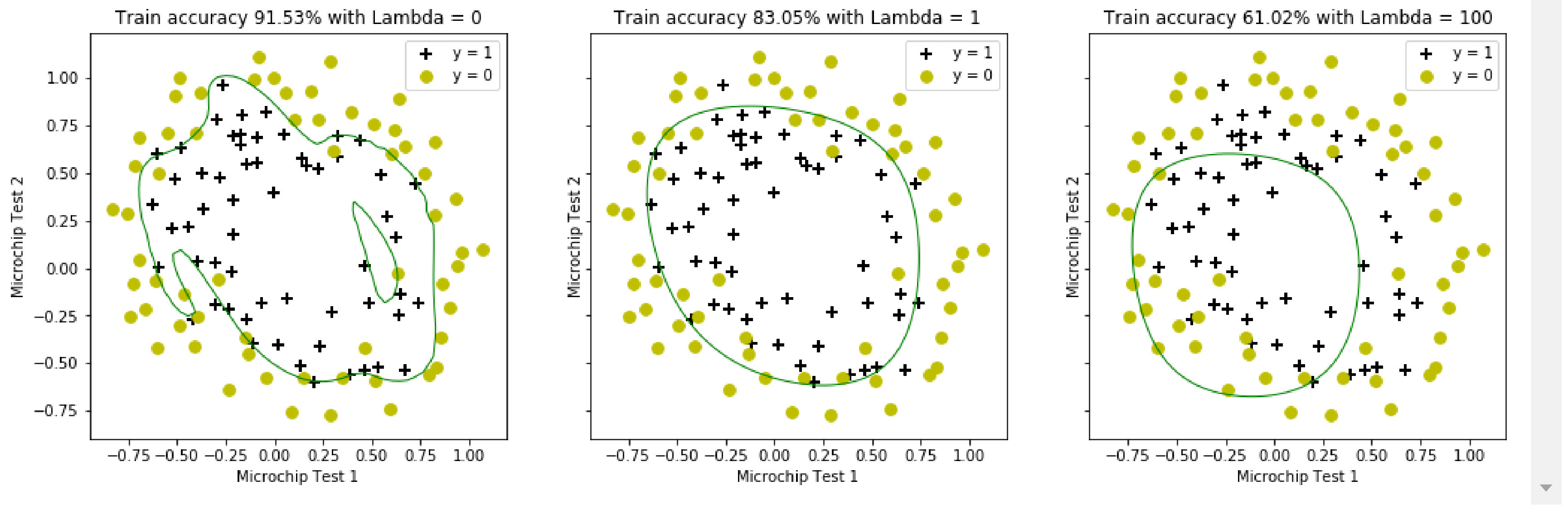
In [14]:
```
def costFunctionReg(theta, reg, *args):
    m = y.size
    h = sigmoid(XX.dot(theta))

    J = -1*(1/m)*(np.log(h).T.dot(y)+np.log(1-h).T.dot(1-y)) + (reg/(2*m))*np.sum(np.square(theta[1:]))

    if np.isnan(J[0]):
        return(np.inf)
    return(J[0])

def gradientReg(theta, reg, *args):
    m = y.size
    h = sigmoid(XX.dot(theta.reshape(-1,1)))

    grad = (1/m)*XX.T.dot(h-y) + (reg/m)*np.r_[[[0]],theta[1:].reshape(-1,1)]

    return(grad.flatten())
```

In [15]:
```
initial_theta = np.zeros(XX.shape[1])
costFunctionReg(initial_theta, 1, XX, y)
```

Out[15]:
```
0.6931471805599453
```

```
In [17]: fig, axes = plt.subplots(1,3, sharey = True, figsize=(17,5))

         # Decision boundaries
         # Lambda = 0 : No regularization --> too flexible, overfitting the training data
         # Lambda = 1 : Looks about right
         # Lambda = 100 : Too much regularization --> high bias

         for i, C in enumerate([0, 1, 100]):
             # Optimize costFunctionReg
             res2 = minimize(costFunctionReg, initial_theta, args=(C, XX, y), method=None, jac=gradientReg, options={'max

             # Accuracy
             accuracy = 100*sum(predict(res2.x, XX) == y.ravel())/y.size

             # Scatter plot of X,y
             plotData(data2, 'Microchip Test 1', 'Microchip Test 2', 'y = 1', 'y = 0', axes.flatten()[i])

             # Plot decisionboundary
             x1_min, x1_max = X[:,0].min(), X[:,0].max(),
             x2_min, x2_max = X[:,1].min(), X[:,1].max(),
             xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max), np.linspace(x2_min, x2_max))
             h = sigmoid(poly.fit_transform(np.c_[xx1.ravel(), xx2.ravel()]).dot(res2.x))
             h = h.reshape(xx1.shape)
             axes.flatten()[i].contour(xx1, xx2, h, [0.5], linewidths=1, colors='g');
             axes.flatten()[i].set_title('Train accuracy {}% with Lambda = {}'.format(np.round(accuracy, decimals=2), C))
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: RuntimeWarning: overflow encountered in exp
```



How logistic regression gives nonlinear decision boundary?

In [45]:
```
import numpy as np
import math
import time
import pandas as pd
import matplotlib.pyplot as plt

X = np.ndfromtxt('images.csv', delimiter=',')
y = np.ndfromtxt("labels.csv", delimiter=',', dtype=np.int8)
img_size = X.shape[1]

#Use another way wo read in the data
XX = pd.read_csv("images.csv",header = None).values
yy = pd.read_csv("labels.csv",header = None, dtype= np.int8).values
print(X.shape,y.shape)
print(yy.shape, y.shape)

y=y.reshape(10000,1) #Original its dim is (10000,) and thus cannot compare with array yy.
print(yy.shape, y.shape)
print(np.array_equal(yy,y)) #Better than print(yy == y) which print out an boolean array.

#After comparison, reshape it back for later use.
y = y.reshape(10000) #y is reshaped to (10000,1) for comparison purpose earlier.

#Using the reshape to flip between 1D and 2D images. It is better than other functions.
X = df.values
plt.imshow(X[300].reshape(28,28), cmap=plt.cm.gray_r, interpolation='nearest')
plt.show()

# To filter out two classes for binary logistic regression
# If the the dimension of y:(10000,) becomes (10000,1), then the following code will  have problem.
ind = np.logical_or(y == 1, y == 0)
ind.shape
X = X[ind, :]
y = y[ind]
print(X.shape,y.shape)

num_train = int(len(y) * 0.8)
X_train = X[0:num_train, :] #This will not include the X[num_train,:]
X_test = X[num_train:-1,:]  #This will not include the final row index.
y_train = y[0:num_train]
y_test = y[num_train:-1]
```

```python
def h1(theta, x):
    sum = 0.0
    for i in range(len(x)):
        sum -= theta[i] * x[i]
    return 1 / (1 + math.exp(sum))

def h2(theta, x):
    return 1 / (1 + np.exp(-np.dot(theta, x))) #minus sign added by me.

theta = np.zeros([img_size])
x = X[0,:]
print(x.shape)
%timeit h1(theta, x)
%timeit h2(theta, x)
#The vectorized version is much faster.

# doing everything element-wise
def h(theta, x):
    return 1 / (1 + np.exp(-np.dot(theta, x)))

def GD_elementwise(theta, X_train, y_train, alpha):
    diff_arr = np.zeros([len(y_train)])
    for m in range(len(y_train)):
            diff_arr[m] = h(theta, X_train[m, :]) - y_train[m]
    for j in range(len(theta)):
        s = 0.0
        for m in range(len(y_train)):
            s += diff_arr[m] * X_train[m, j]
        theta[j] = theta[j] - alpha * s

def train_elementwise(X_train, y_train, max_iter, alpha):
    theta = np.zeros([img_size]) #In neural network, we cannot set parameters to zero.
    for i in range(max_iter):
        GD_elementwise(theta, X_train, y_train, alpha)
    return theta

max_iter = 10
alpha = 0.01
start = time.time()
theta = train_elementwise(X_train, y_train, max_iter, alpha)
end = time.time()
print(theta.shape)
# print("time elapsed: {0} seconds".format(end - start))
# pred = (np.sign(h(theta, X_test) - 0.5) + 1) / 2
# print("percentage correct: {0}".format(np.sum(pred == y_test) / len(y_test)))

#some vectorization
def h_vec(theta, X):
    return 1 / (1 + np.exp(-np.matmul(X, theta)))

def GD_better(theta,  X_train, y_train, alpha):
    diff_arr = h_vec(theta, X_train) - y_train
    for j in range(len(theta)):
        theta[j] = theta[j] - alpha * np.dot(diff_arr, X_train[:, j])

def train_better(X_train, y_train, max_iter, alpha):
    theta = np.zeros([img_size])
    for i in range(max_iter):
        GD_better(theta, X_train, y_train, alpha)
    return theta

max_iter = 10
alpha = 0.01
start = time.time()
theta = train_better(X_train, y_train, max_iter, alpha)
end = time.time()
print("time elapsed: {0} seconds".format(end - start))
pred = (np.sign(h_vec(theta, X_test) - 0.5) + 1) / 2
print("percentage correct: {0}".format(np.sum(pred == y_test) / len(y_test)))
print(sum(pred==y_test))

#fully vectorized
def GD (theta, X_train, y_train, alpha):
    theta -= alpha * np.squeeze(np.matmul(np.reshape(h_vec(theta, X_train) - y_train, [1, -1]), X_train))

def train_vec(X_train, y_train, max_iter, alpha):
    theta = np.zeros([img_size])
    for i in range(max_iter):
        GD(theta, X_train, y_train, alpha)
    return theta

max_iter = 10
alpha = 0.01
start = time.time()
theta = train_vec(X_train, y_train, max_iter, alpha)
end = time.time()
```

```
print("time elapsed: {0} seconds".format(end - start))
pred = (np.sign(h_vec(theta, X_test) - 0.5) + 1) / 2
print("percentage correct: {0}".format(np.sum(pred == y_test) / len(y_test)))
```

```
(10000, 784) (10000,)
(10000, 1) (10000,)
(10000, 1) (10000, 1)
True

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-45-3bac6d8f206a> in <module>()
     23 
     24 #Using the reshape to flip between 1D and 2D images. It is better than other functions.
---> 25 X = df.values
     26 plt.imshow(X[300].reshape(28,28), cmap=plt.cm.gray_r, interpolation='nearest')
     27 plt.show()

NameError: name 'df' is not defined
```

Notice how the cross-validation scores change with different alphas. Which alpha should you pick? How can you fine-tune your model? You'll learn all about this in the next chapter!

Exercise 1 Diabetes classification

The PIMA Indians dataset is used to predict whether or not a given female patient will contract diabetes.

Metrics for classification

In [18]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
import pandas as pd

filePath = "C:/Users/ljyan/Desktop/courseNotes/dataScience/machineLearning/data/"
filename = "diabetes.csv"
file = filePath+filename
df = pd.read_csv(file, sep =  ',')

X = df.drop('diabetes',axis = 1).values
y = df['diabetes'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.4, random_state=42)
logreg = LogisticRegression(solver = 'lbfgs',max_iter = 300)
logreg.fit(X_train, y_train)
y_pred = logreg.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

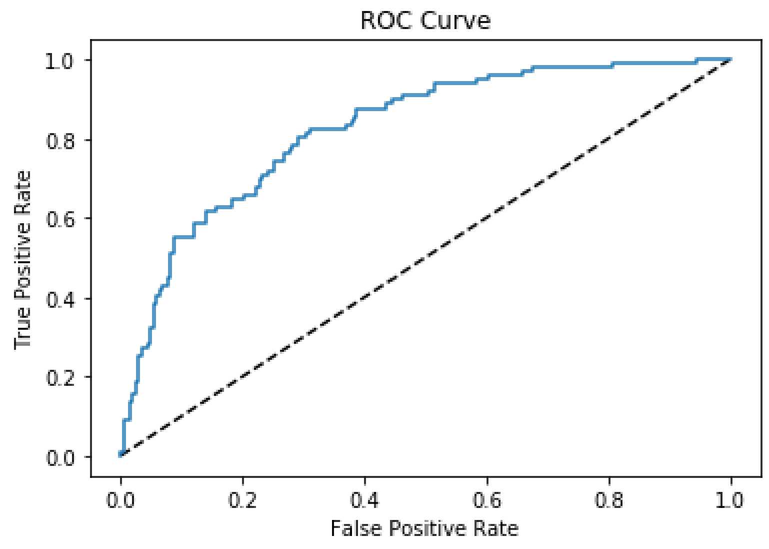
```
[[168  38]
 [ 36  66]]
              precision    recall  f1-score   support

           0       0.82      0.82      0.82       206
           1       0.63      0.65      0.64       102

   micro avg       0.76      0.76      0.76       308
   macro avg       0.73      0.73      0.73       308
weighted avg       0.76      0.76      0.76       308
```

Plotting an ROC curve

Classification reports and confusion matrices are great methods to quantitatively evaluate model performance, while ROC curves provide a way to visually evaluate models. Most classifiers in scikit-learn have a .predict_proba() method which returns the probability of a given sample being in a particular class. Here the returns of .predict_proba() will be used to plot ROC curve.

```
In [20]:
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve
y_pred_prob = logreg.predict_proba(X_test)[:,1]
# type(y_pred_prob) is ndarray, shape is (308,2),The first column is prob, and second is 1-prob.

# Generate ROC curve values: fpr, tpr, thresholds
fpr, tpr, thresholds = roc_curve(y_test,y_pred_prob)
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
```

```
Out[20]: Text(0.5,1,'ROC Curve')
```



AUC computation

Random guesses would be correct approximately 50% of the time and the resulting ROC curve would be a diagonal line in which the True Positive Rate and False Positive Rate are always equal. The Area under this ROC curve would be 0.5. If the AUC is greater than 0.5, the model is better than random guessing.

In [21]:
```
# Import necessary modules
from sklearn.model_selection import cross_val_score
from sklearn.metrics import roc_auc_score

y_pred_prob = logreg.predict_proba(X_test)[:,1]

print("AUC: {}".format(roc_auc_score(y_test, y_pred_prob)))

# Compute cross-validated AUC scores: cv_auc
cv_auc = cross_val_score(logreg, X, y, cv=5, scoring='roc_auc')

print("AUC scores computed using 5-fold cross-validation: {}".format(cv_auc))
```

```
AUC: 0.8242908814011042
AUC scores computed using 5-fold cross-validation: [0.81240741 0.80777778 0.82555556 0.87283019 0.84471698]
```

Hyperparameter tuning with GridSearchCV

Like the alpha parameter of lasso and ridge regularization that you saw earlier, in scikit learn logistic regression also has a regularization parameter: C. C controls the **inverse of the regularization strength**. A large C can lead to an overfit model, while a small C can lead to an underfit model.

In [23]:
```python
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
import numpy as np

# Setup the hyperparameter grid for C
c_space = np.logspace(-5, 8, 15)
param_grid = {'C': c_space}

logreg = LogisticRegression(solver = 'lbfgs', max_iter=300)
#solver = 'lbfgs' is usef for silencing a lot of warning.
#max_iter = 300 (default is 100) is used for convergence.
logreg_cv = GridSearchCV(logreg, param_grid, cv=5)

logreg_cv.fit(X,y)

print("Tuned Logistic Regression Parameters: {}".format(logreg_cv.best_params_))
print("Best score is {}".format(logreg_cv.best_score_))
```

```
Tuned Logistic Regression Parameters: {'C': 0.006105402296585327}
Best score is 0.7734375
```

Evaluating tuned hyperparameters on a hold-out set

In addition to C, logistic regression has a 'penalty' hyperparameter which specifies whether to use 'l1' or 'l2' regularization.

In [25]:
```python
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
import numpy as np

c_space = np.logspace(-5, 8, 15)
param_grid = {'C': c_space, 'penalty': ['l1', 'l2']}
# param_grid = {'C': c_space, 'penalty': ['l2']} # solver = 'lbfgs' support only l2

logreg = LogisticRegression(solver = 'liblinear', max_iter=300)

X_train, X_test, y_train, y_test = train_test_split(X,y,test_size = 0.4,random_state = 42)

logreg_cv = GridSearchCV(logreg,param_grid,cv=5)

logreg_cv.fit(X_train,y_train)

# Not work on the hold-out set yet.

print("Tuned Logistic Regression Parameter: {}".format(logreg_cv.best_params_))
print("Tuned Logistic Regression Accuracy: {}".format(logreg_cv.best_score_))
```

```
Tuned Logistic Regression Parameter: {'C': 31.622776601683793, 'penalty': 'l2'}
Tuned Logistic Regression Accuracy: 0.7673913043478261

C:\Users\ljyan\Anaconda3\lib\site-packages\sklearn\model_selection\_search.py:841: DeprecationWarning: The defa
ult of the `iid` parameter will change from True to False in version 0.22 and will be removed in 0.24. This wil
l change numeric results when test-set sizes are unequal.
  DeprecationWarning)
```