Logistic Regression

- 建立一个逻辑回归模型来预测一个学生是否被大学录取。
- 假设一个大学系的管理员，根据两次考试的结果来决定每个申请人的录取机会。
- 有以前的申请人的历史数据
- 有两个考试的申请人的分数和录取决定。
- 建立一个分类模型，根据考试成绩估计入学概率。

In [1]:

```
#三大件
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:

```
import os
# path = 'data' + os.sep + 'LogiReg_data.txt'
# 此处我把数据文件直接放在与编码的同个文件夹下，因此
path = 'LogiReg_data.txt'

pdData = pd.read_csv(path, header=None, names=['Exam 1', 'Exam 2', 'Admitted'])
# 此处需要给读取的CSV加表头，否则读取时会以第一行数据作为表头
# pdData = pd.read_csv(path)
pdData.head()
```

Out[2]:

	Exam 1	Exam 2	Admitted
0	34.623660	78.024693	0
1	30.286711	43.894998	0
2	35.847409	72.902198	0
3	60.182599	86.308552	1
4	79.032736	75.344376	1

python中os.path常用模块

- os.path.sep:路径分隔符 linux下就用这个了'/'
- os.path.altsep: 根目录
- os.path.curdir:当前目录
- os.path.pardir：父目录
- os.path.abspath(path)：绝对路径
- os.path.join(): 常用来链接路径
- os.path.split(path): 把path分为目录和文件两个部分，以列表返回

In [3]:

```
print ("os.path.sep:"+os.path.sep)
print ("os.path.altsep:"+os.path.altsep)
print ("os.path.curdir:"+os.path.curdir)
print ("os.path.pardir:"+os.path.pardir)
print ("os.path.abspath(path):",os.path.abspath(path))
print ("os.path.join(path):",os.path.join(path))
print ("os.path.split(path):",os.path.split(path))
```

```
os.path.sep:\
os.path.altsep:/
os.path.curdir:.
os.path.pardir:..
os.path.abspath(path): C:\Users\许晴雯\01_Python\梯度下降求解逻辑回归\LogiReg_data.t
xt
os.path.join(path): LogiReg_data.txt
os.path.split(path): ('', 'LogiReg_data.txt')
```

In [4]:

```
# 例牌，查看数据维度
pdData.shape
```

Out[4]:

```
(100, 3)
```

In [5]:

```
positive = pdData[pdData['Admitted'] == 1] # returns the subset of rows such Admitted = 1, i.e. the
print(positive.head())
```

```
       Exam 1     Exam 2  Admitted
3   60.182599  86.308552         1
4   79.032736  75.344376         1
6   61.106665  96.511426         1
7   75.024746  46.554014         1
8   76.098787  87.420570         1
```

In [6]:

```
negative = pdData[pdData['Admitted'] == 0] # returns the subset of rows such Admitted = 0, i.e. the
print(negative.head())
```

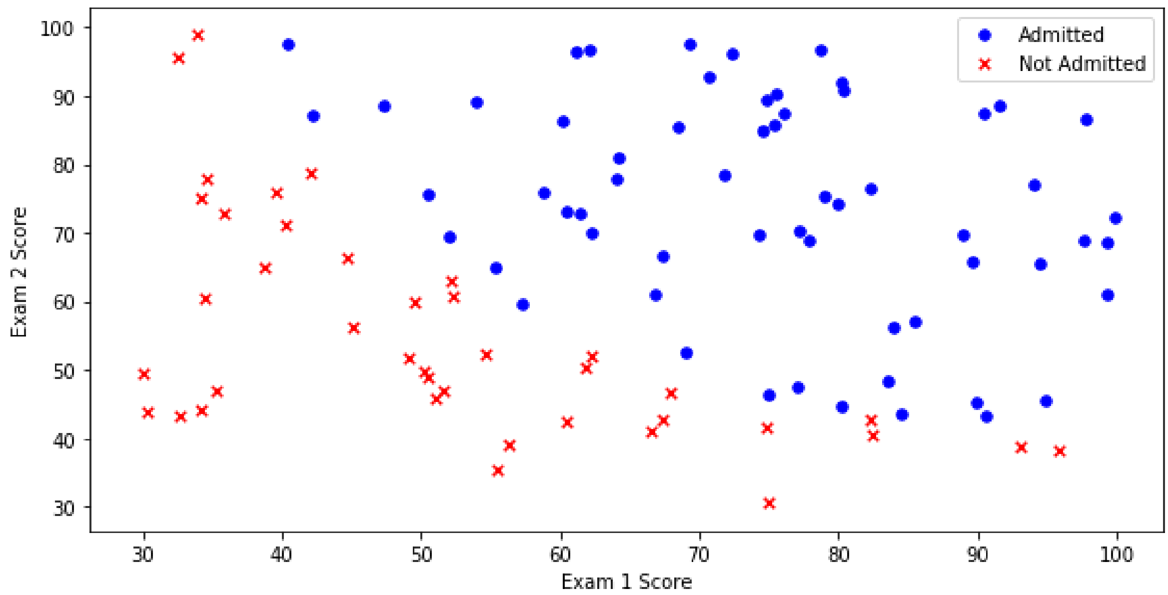
```
        Exam 1     Exam 2  Admitted
0    34.623660  78.024693         0
1    30.286711  43.894998         0
2    35.847409  72.902198         0
5    45.083277  56.316372         0
10   95.861555  38.225278         0
```

In [7]:

```
# fig for figure (数据) , ax for axes (轴)
fig, ax = plt.subplots(figsize=(10,5))
ax.scatter(positive['Exam 1'], positive['Exam 2'], s=30, c='b', marker='o', label='Admitted')
ax.scatter(negative['Exam 1'], negative['Exam 2'], s=30, c='r', marker='x', label='Not Admitted')
ax.legend()
ax.set_xlabel('Exam 1 Score')
ax.set_ylabel('Exam 2 Score')
```

Out[7]:

<matplotlib.text.Text at 0x19ee24246a0>



The logistic regression

目标：建立分类器（求解出三个参数 $\theta_0 \theta_1 \theta_2$ ）

设定阈值，根据阈值判断录取结果

要完成的模块

- `sigmoid` : 映射到概率的函数
- `model` : 返回预测结果值
- `cost` : 根据参数计算损失
- `gradient` : 计算每个参数的梯度方向
- `descent` : 进行参数更新
- `accuracy`: 计算精度

`sigmoid` 函数

$$g(z) = \frac{1}{1 + e^{-z}}$$

- 将任意的输入映射到了[0,1]区间

- 我们在线性回归中可以得到一个预测值，再将该值映射到Sigmoid 函数中
- 这样就完成了由值到概率的转换，也就是分类任务

In [8]:

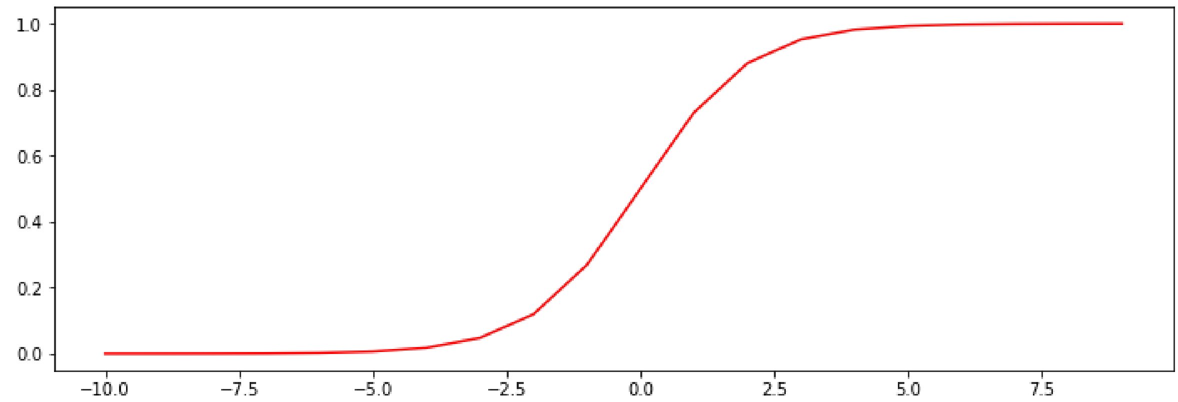
```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

In [9]:

```
nums = np.arange(-10, 10, step=1) #creates a vector containing 20 equally spaced values from -10 to
fig, ax = plt.subplots(figsize=(12,4))
ax.plot(nums, sigmoid(nums), 'r')
```

Out[9]:

```
[<matplotlib.lines.Line2D at 0x19ee23d5860>]
```



help(plt.subplots)

Sigmoid

- $g: \mathbb{R} \to [0,1]$
- $g(0) = 0.5$
- $g(-\infty) = 0$
- $g(+\infty) = 1$

In [10]:

```
def model(X, theta):

    return sigmoid(np.dot(X, theta.T))
```

$$\begin{pmatrix}\theta_0 & \theta_1 & \theta_2\end{pmatrix} \times \begin{pmatrix}1\\ x_1\\ x_2\end{pmatrix} = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

In [11]:

```
# 在最前面插入一列元素均为1的向量，代表即将与θ0相乘的X1=1
pdData.insert(0, 'Ones', 1) # in a try / except structure so as not to return an error if the block

# set X (training data) and y (target variable)
# convert the Pandas representation of the data to an array useful for further computations
orig_data = pdData.as_matrix()
cols = orig_data.shape[1]
X = orig_data[:,0:cols-1]
y = orig_data[:,cols-1:cols]

# convert to numpy arrays and initalize the parameter array theta
#X = np.matrix(X.values)
#y = np.matrix(data.iloc[:,3:4].values) #np.array(y.values)

# 定义θ是一个1*3的向量
theta = np.zeros([1, 3])
```

In [12]:

```
orig_data[1:5,]
```

Out[12]:

```
array([[ 1.        , 30.28671077, 43.89499752,  0.        ],
       [ 1.        , 35.84740877, 72.90219803,  0.        ],
       [ 1.        , 60.18259939, 86.3085521 ,  1.        ],
       [ 1.        , 79.03273605, 75.34437644,  1.        ]])
```

In [13]:

```
X[:5]
```

Out[13]:

```
array([[ 1.        , 34.62365962, 78.02469282],
       [ 1.        , 30.28671077, 43.89499752],
       [ 1.        , 35.84740877, 72.90219803],
       [ 1.        , 60.18259939, 86.3085521 ],
       [ 1.        , 79.03273605, 75.34437644]])
```

In [14]:

```
y[:5]
```

Out[14]:

```
array([[0.],
       [0.],
       [0.],
       [1.],
       [1.]])
```

In [15]:

```
theta
```

Out[15]:

```
array([[0., 0., 0.]])
```

In [16]:

```
X.shape, y.shape, theta.shape
```

Out[16]:

((100, 3), (100, 1), (1, 3))

损失函数

将对数似然函数去负号

$$D(h_\theta(x), y) = -y\log(h_\theta(x)) - (1-y)\log(1-h_\theta(x))$$

求平均损失

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} D(h_\theta(x_i), y_i)$$

In [17]:

```
def cost(X, y, theta):
    left = np.multiply(-y, np.log(model(X, theta)))
    right = np.multiply(1 - y, np.log(1 - model(X, theta)))
    return np.sum(left - right) / (len(X))
```

In [18]:

```
cost(X, y, theta)
```

Out[18]:

0.6931471805599453

计算梯度

$$\frac{\partial J}{\partial \theta_j} = -\frac{1}{m}\sum_{i=1}^{n}(y_i - h_\theta(x_i))x_{ij}$$

In [19]:

```
def gradient(X, y, theta):
    grad = np.zeros(theta.shape)
    error = (model(X, theta)- y).ravel()
    for j in range(len(theta.ravel())): #for each parmeter
        term = np.multiply(error, X[:,j])
        grad[0, j] = np.sum(term) / len(X)

    return grad
```

Gradient descent

比较3种不同梯度下降方法

In [20]:

```
STOP_ITER = 0
STOP_COST = 1
STOP_GRAD = 2

def stopCriterion(type, value, threshold):
    #设定三种不同的停止策略
    if type == STOP_ITER:        return value > threshold
    elif type == STOP_COST:      return abs(value[-1]-value[-2]) < threshold
    elif type == STOP_GRAD:      return np.linalg.norm(value) < threshold
```

In [21]:

```
import numpy.random
#洗牌
def shuffleData(data):
    np.random.shuffle(data)
    cols = data.shape[1]
    X = data[:, 0:cols-1]
    y = data[:, cols-1:]
    return X, y
```

In [22]:

```
import time

def descent(data, theta, batchSize, stopType, thresh, alpha):
    #梯度下降求解

    init_time = time.time()
    i = 0 # 迭代次数
    k = 0 # batch
    X, y = shuffleData(data)
    grad = np.zeros(theta.shape) # 计算的梯度
    costs = [cost(X, y, theta)] # 损失值


    while True:
        grad = gradient(X[k:k+batchSize], y[k:k+batchSize], theta)
        k += batchSize #取batch数量个数据
        if k >= n:
            k = 0
            X, y = shuffleData(data) #重新洗牌
        theta = theta - alpha*grad # 参数更新
        costs.append(cost(X, y, theta)) # 计算新的损失
        i += 1

        if stopType == STOP_ITER:       value = i
        elif stopType == STOP_COST:     value = costs
        elif stopType == STOP_GRAD:     value = grad
        if stopCriterion(stopType, value, thresh): break

    return theta, i-1, costs, grad, time.time() - init_time
```

In [23]:

```
def runExpe(data, theta, batchSize, stopType, thresh, alpha):
    #import pdb; pdb.set_trace();
    theta, iter, costs, grad, dur = descent(data, theta, batchSize, stopType, thresh, alpha)
    name = "Original" if (data[:,1]>2).sum() > 1 else "Scaled"
    name += " data - learning rate: {} - ".format(alpha)
    if batchSize==n: strDescType = "Gradient"
    elif batchSize==1:  strDescType = "Stochastic"
    else: strDescType = "Mini-batch ({})".format(batchSize)
    name += strDescType + " descent - Stop: "
    if stopType == STOP_ITER: strStop = "{} iterations".format(thresh)
    elif stopType == STOP_COST: strStop = "costs change < {}".format(thresh)
    else: strStop = "gradient norm < {}".format(thresh)
    name += strStop
    print ("***{}\nTheta: {} - Iter: {} - Last cost: {:03.2f} - Duration: {:03.2f}s".format(
        name, theta, iter, costs[-1], dur))
    fig, ax = plt.subplots(figsize=(12,4))
    ax.plot(np.arange(len(costs)), costs, 'r')
    ax.set_xlabel('Iterations')
    ax.set_ylabel('Cost')
    ax.set_title(name.upper() + ' - Error vs. Iteration')
    return theta
```

不同的停止策略

设定迭代次数

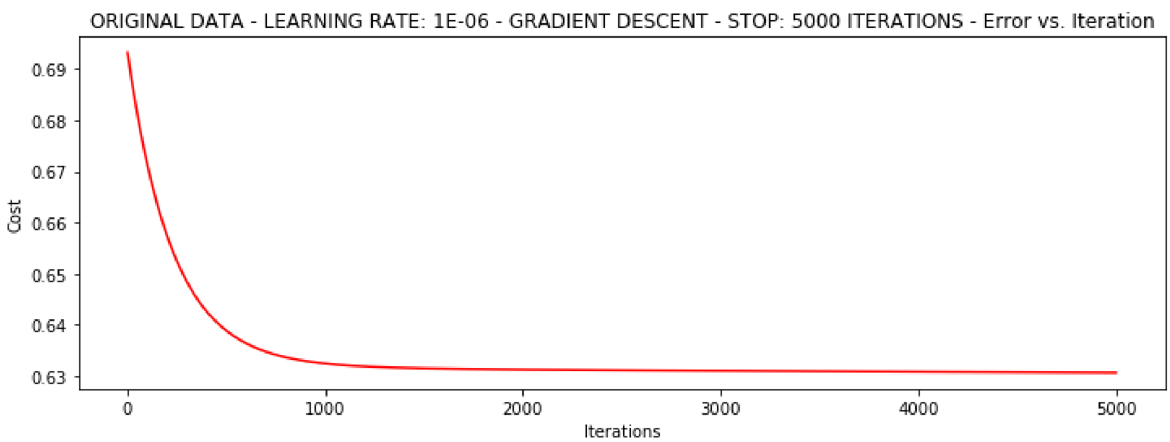
In [24]:

```
#选择的梯度下降方法是基于所有样本的
n=100
runExpe(orig_data, theta, n, STOP_ITER, thresh=5000, alpha=0.000001)
```

```
***Original data - learning rate: 1e-06 - Gradient descent - Stop: 5000 iterations
Theta: [[-0.00027127  0.00705232  0.00376711]] - Iter: 5000 - Last cost: 0.63 - Dura
tion: 1.50s
```

Out[24]:

```
array([[-0.00027127,  0.00705232,  0.00376711]])
```



根据损失值停止

设定阈值 1E-6, 差不多需要110 000次迭代

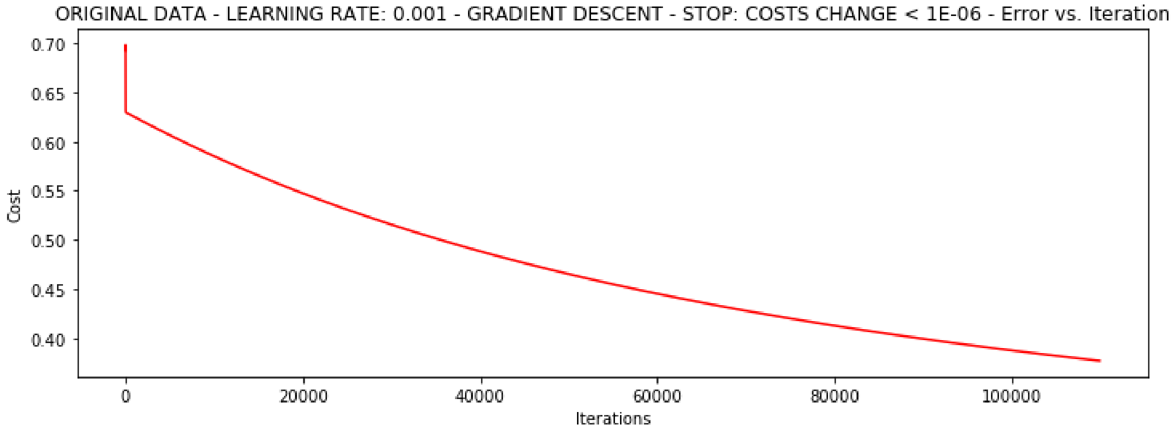
In [25]:

```
runExpe(orig_data, theta, n, STOP_COST, thresh=0.000001, alpha=0.001)
```

```
***Original data - learning rate: 0.001 - Gradient descent - Stop: costs change < 1e
-06
Theta: [[-5.13364014  0.04771429  0.04072397]] - Iter: 109901 - Last cost: 0.38 - Du
ration: 32.95s
```

Out[25]:

```
array([[-5.13364014,  0.04771429,  0.04072397]])
```



根据梯度变化停止

设定阈值 0.05,差不多需要40 000次迭代

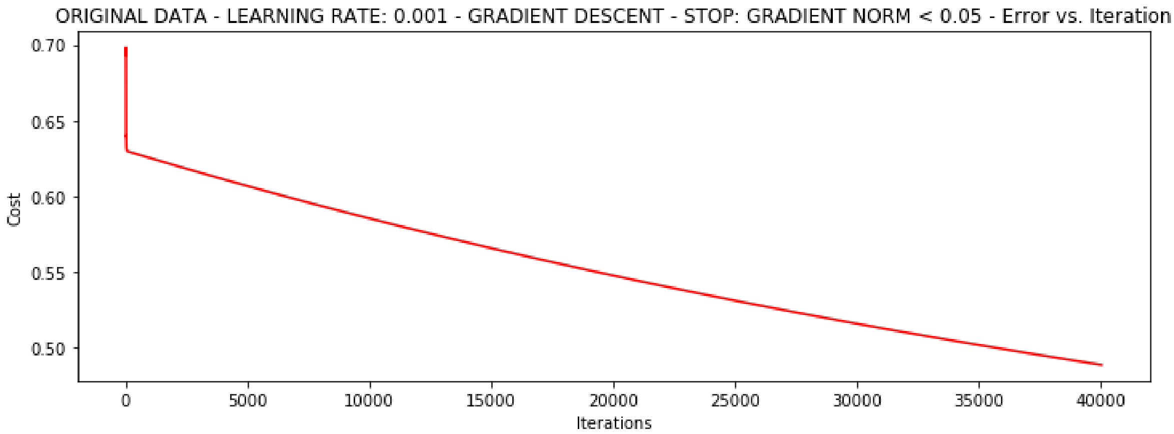
In [26]:

```
runExpe(orig_data, theta, n, STOP_GRAD, thresh=0.05, alpha=0.001)
```

```
***Original data - learning rate: 0.001 - Gradient descent - Stop: gradient norm <
 0.05
Theta: [[-2.37033409  0.02721692  0.01899456]] - Iter: 40045 - Last cost: 0.49 - Dur
ation: 12.52s
```

Out[26]:

```
array([[-2.37033409,  0.02721692,  0.01899456]])
```



对比不同的梯度下降方法

Stochastic descent

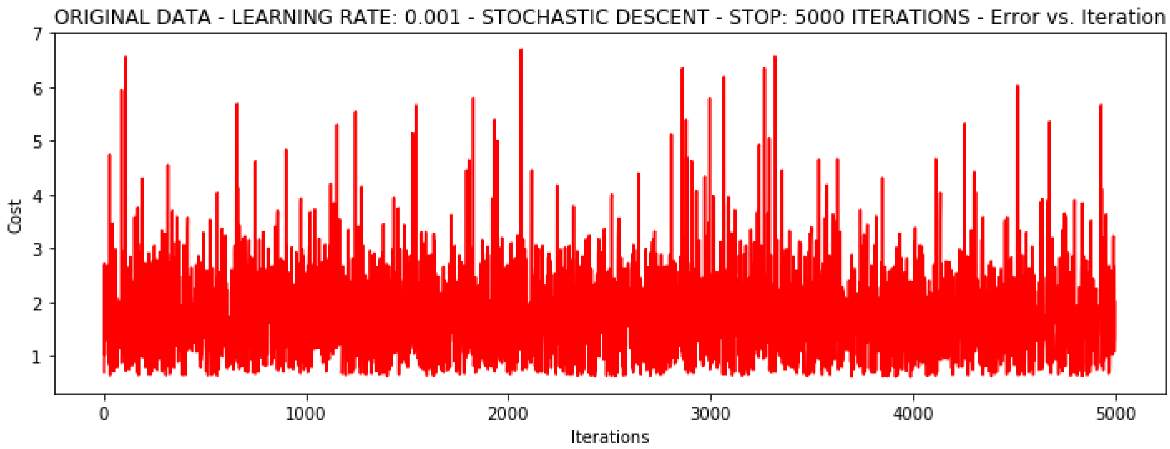
In [27]:

```
runExpe(orig_data, theta, 1, STOP_ITER, thresh=5000, alpha=0.001)
```

```
***Original data - learning rate: 0.001 - Stochastic descent - Stop: 5000 iterations
Theta: [[-0.38572858  0.06276256 -0.09757208]] - Iter: 5000 - Last cost: 2.01 - Dura
tion: 0.52s
```

Out[27]:

```
array([[-0.38572858,  0.06276256, -0.09757208]])
```



有点爆炸。。。很不稳定,再来试试把学习率调小一些

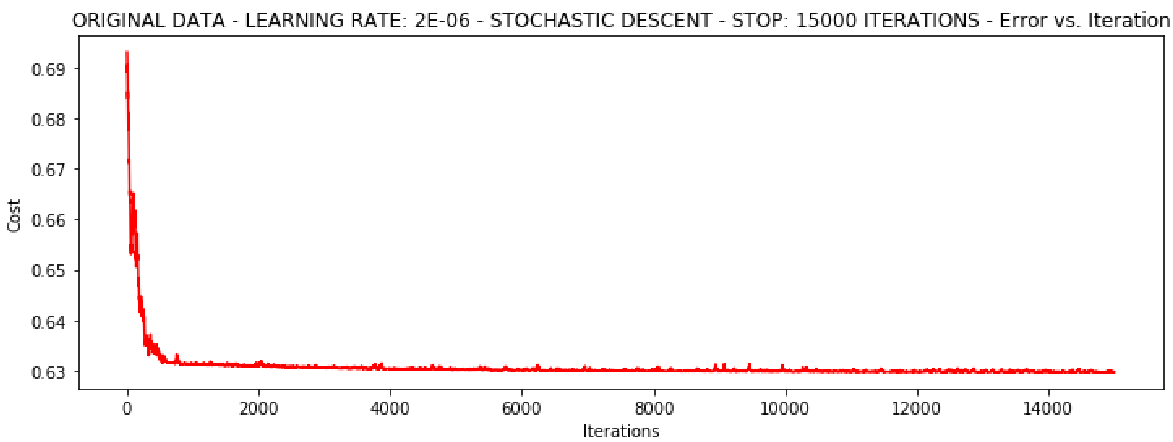
In [28]:

```
runExpe(orig_data, theta, 1, STOP_ITER, thresh=15000, alpha=0.000002)
```

```
***Original data - learning rate: 2e-06 - Stochastic descent - Stop: 15000 iteration
s
Theta: [[-0.00202143  0.01003913  0.00096006]] - Iter: 15000 - Last cost: 0.63 - Dur
ation: 1.55s
```

Out[28]:

```
array([[-0.00202143,  0.01003913,  0.00096006]])
```



速度快，但稳定性差，需要很小的学习率

Mini-batch descent

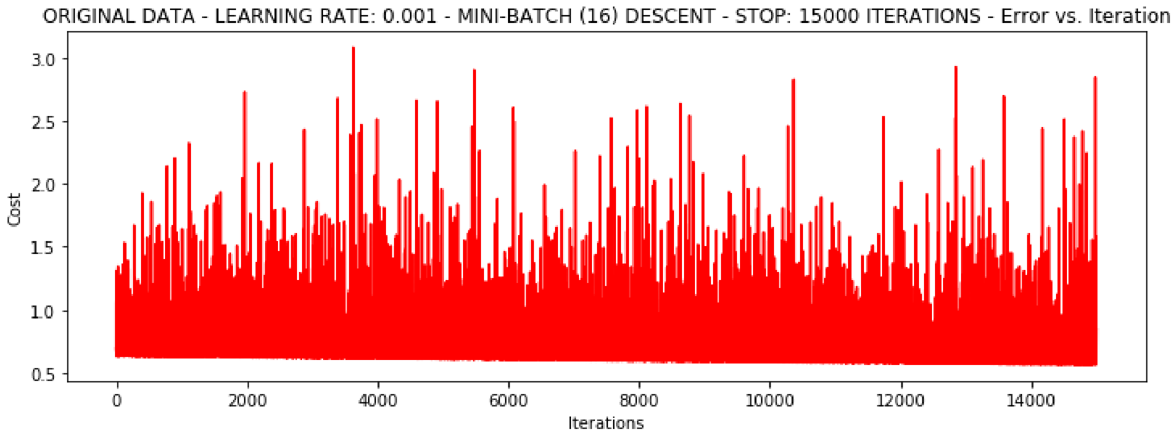
In [29]:

```
runExpe(orig_data, theta, 16, STOP_ITER, thresh=15000, alpha=0.001)
```

```
***Original data - learning rate: 0.001 - Mini-batch (16) descent - Stop: 15000 iter
ations
Theta: [[-1.03674852e+00  2.89117689e-03  1.62927760e-04]] - Iter: 15000 - Last cos
t: 0.85 - Duration: 1.96s
```

Out[29]:

```
array([[-1.03674852e+00,  2.89117689e-03,  1.62927760e-04]])
```



浮动仍然比较大，我们来尝试下对数据进行标准化 将数据按其属性(按列进行)减去其均值，然后除以其方差。最后得到的结果是，对每个属性/每列来说所有数据都聚集在0附近，方差值为1

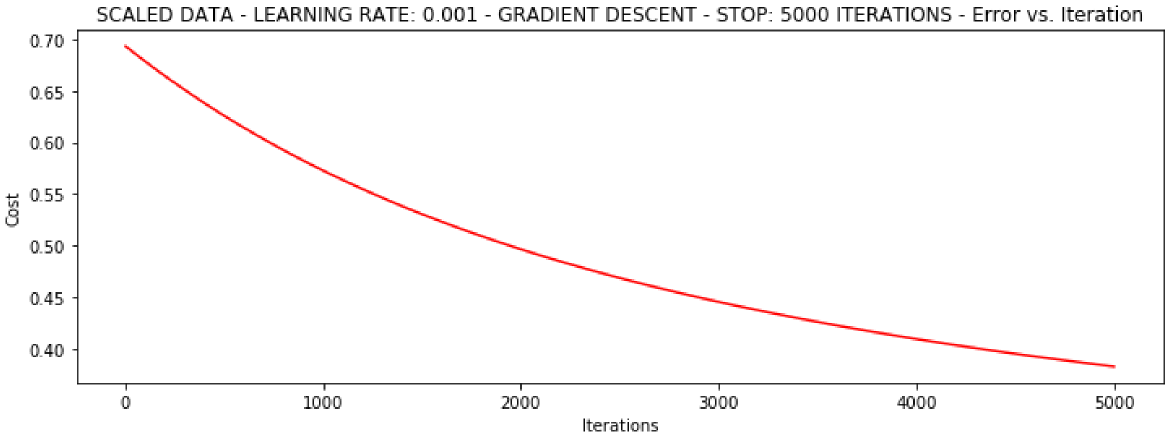
In [30]:

```
from sklearn import preprocessing as pp

scaled_data = orig_data.copy()
scaled_data[:, 1:3] = pp.scale(orig_data[:, 1:3])

runExpe(scaled_data, theta, n, STOP_ITER, thresh=5000, alpha=0.001)
```

```
***Scaled data - learning rate: 0.001 - Gradient descent - Stop: 5000 iterations
Theta: [[0.3080807  0.86494967 0.77367651]] - Iter: 5000 - Last cost: 0.38 - Duratio
n: 1.52s
```

Out[30]:

```
array([[0.3080807 , 0.86494967, 0.77367651]])
```



它好多了！原始数据，只能达到达到0.61，而我们得到了0.38个在这里！ 所以对数据做预处理是非常重要的

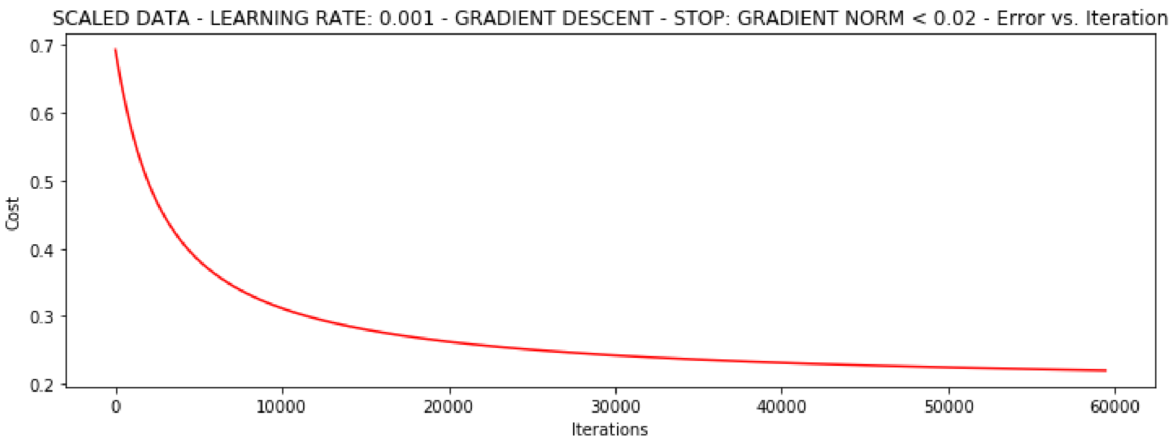
In [31]:

```
runExpe(scaled_data, theta, n, STOP_GRAD, thresh=0.02, alpha=0.001)
```

```
***Scaled data - learning rate: 0.001 - Gradient descent - Stop: gradient norm < 0.0
2
Theta: [[1.0707921  2.63030842 2.41079787]] - Iter: 59422 - Last cost: 0.22 - Durati
on: 19.30s
```

Out[31]:

```
array([[1.0707921 , 2.63030842, 2.41079787]])
```

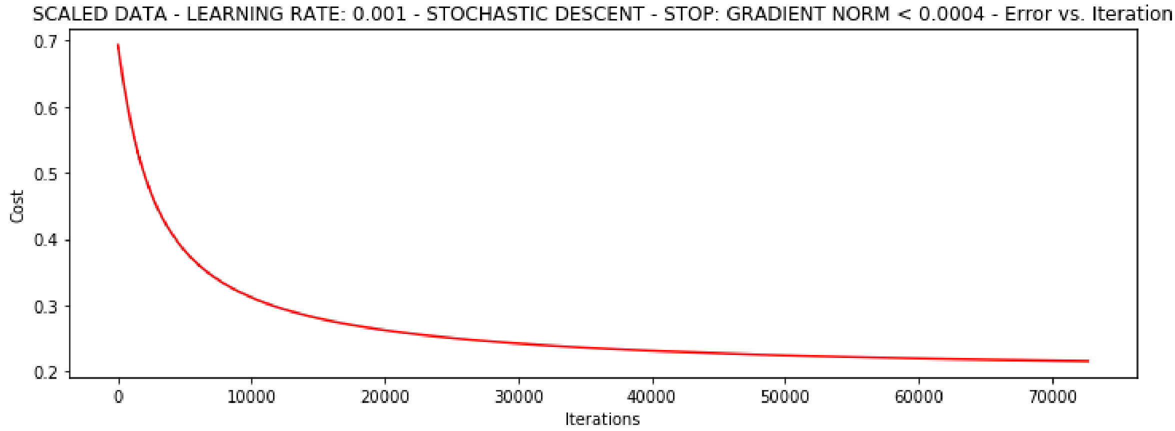


更多的迭代次数会使得损失下降的更多！

In [32]:

```
theta = runExpe(scaled_data, theta, 1, STOP_GRAD, thresh=0.002/5, alpha=0.001)
```

```
***Scaled data - learning rate: 0.001 - Stochastic descent - Stop: gradient norm <
 0.0004
Theta: [[1.14794259 2.79312581 2.56778582]] - Iter: 72674 - Last cost: 0.22 - Durati
on: 9.15s
```



随机梯度下降更快，但是我们需要迭代的次数也需要更多，所以还是用batch的比较合适！！！

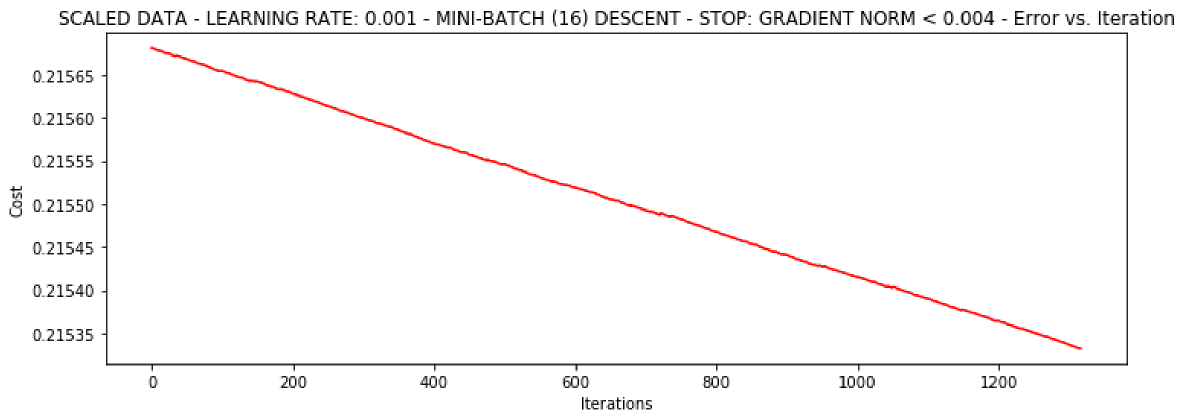
In [33]:

```
runExpe(scaled_data, theta, 16, STOP_GRAD, thresh=0.002*2, alpha=0.001)
```

```
***Scaled data - learning rate: 0.001 - Mini-batch (16) descent - Stop: gradient nor
m < 0.004
Theta: [[1.15645896 2.80696538 2.58218377]] - Iter: 1315 - Last cost: 0.22 - Duratio
n: 0.23s
```

Out[33]:

```
array([[1.15645896, 2.80696538, 2.58218377]])
```



精度

In [34]:

```
#设定阈值
def predict(X, theta):
    return [1 if x >= 0.5 else 0 for x in model(X, theta)]
```