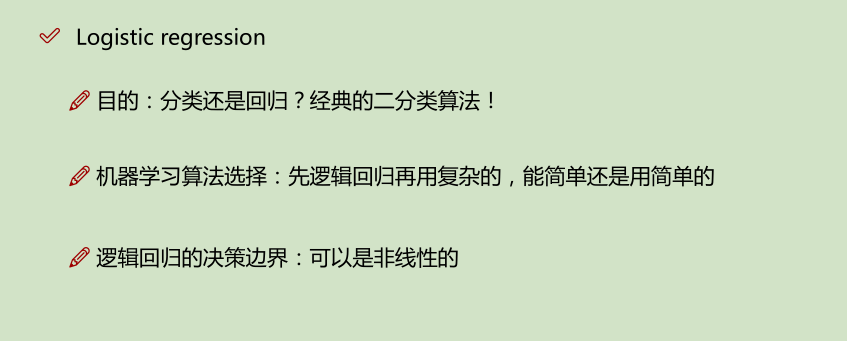
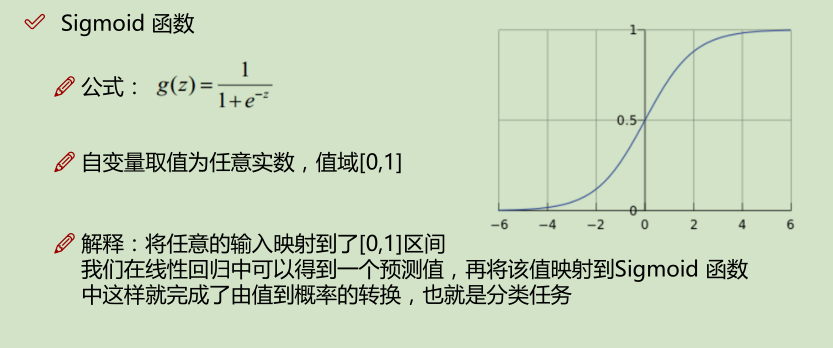
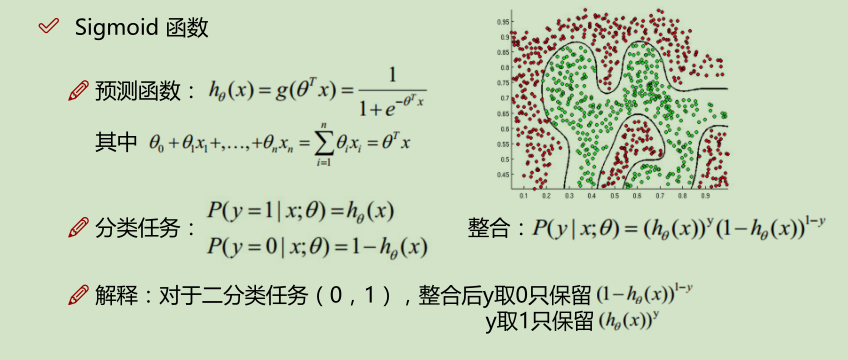
https://www.cnblogs.com/douzujun/p/8370993.html

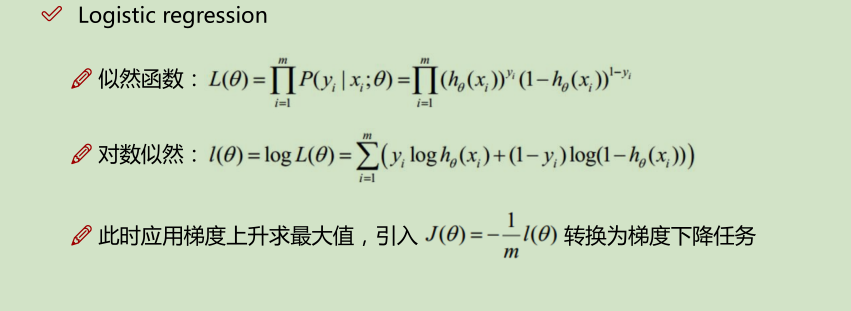
[**机器学习算法整理（二）梯度下降求解逻辑回归 python实现**](http://www.cnblogs.com/douzujun/p/8370993.html)

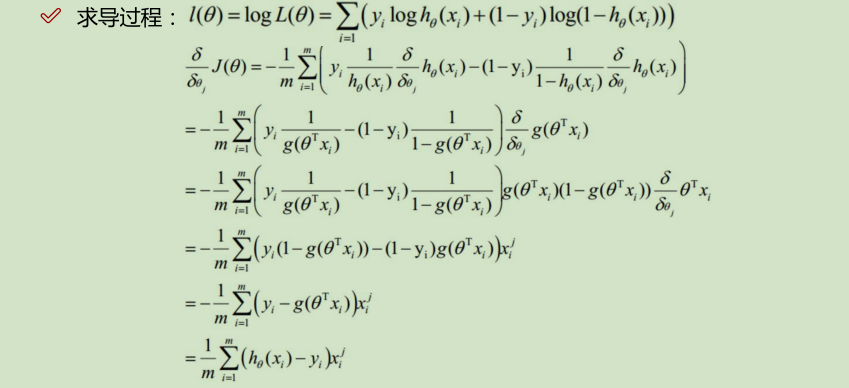
逻辑回归(Logistic regression)

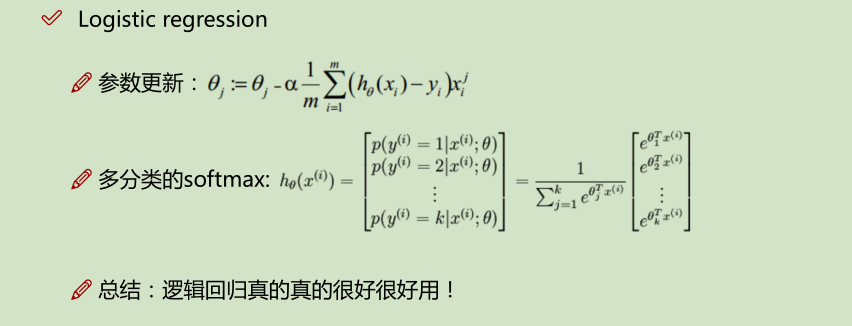












**用梯度下降求解逻辑回归 Logistic Regression**

**The data**

我们将建立一个逻辑回归模型来预测一个学生是否被大学录取。假设你是一个大学系的管理员，你想根据两次考试的结果来决定每个申请人的录取机会。你有以前的申请人的历史数据，你可以用它作为逻辑回归的训练集。对于每一个培训例子，你有两个考试的申请人的分数和录取决定。为了做到这一点，我们将建立一个分类模型，根据考试成绩估计入学概率。

In [7]:

**#三大件**

**import numpy as np**

**import pandas as pd**

**import matplotlib.pyplot as plt**

**%matplotlib inline**

In [8]:

**import os**

**path = 'data' + os.sep + 'LogiReg\_data.txt'**

**pdData = pd.read\_csv(path, header=None, names=['Exam 1', 'Exam 2', 'Admitted'])**

**pdData.head()**

Out[8]:

|  | **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 |

In [10]:

[复制代码](javascript:void(0);)

**positive = pdData[pdData['Admitted'] == 1] # returns the subset of rows such Admitted = 1, i.e. the set of \*positive\* examples**

**negative = pdData[pdData['Admitted'] == 0] # returns the subset of rows such Admitted = 0, i.e. the set of \*negative\* examples**

**​**

**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')**

[复制代码](javascript:void(0);)

Out[10]:

**Text(0,0.5,'Exam 2 Score')**

**The logistic regression**

目标：建立分类器（求解出三个参数 θ0θ1θ2）

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**

https://images2017.cnblogs.com/blog/817161/201802/817161-20180208000059685-826708517.png

In [11]:

**def sigmoid(z):**

**return 1 / (1 + np.exp(-z))**

In [12]:

**nums = np.arange(-10, 10, step=1) #creates a vector containing 20 equally spaced values from -10 to 10**

**fig, ax = plt.subplots(figsize=(12,4))**

**ax.plot(nums, sigmoid(nums), 'r')**

Out[12]:

[<matplotlib.lines.Line2D at 0x15117048>]

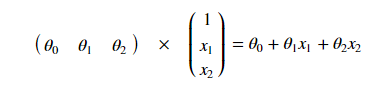
**Sigmoid**

* g:ℝ→[0,1]
* g(0)=0.5
* g(−∞)=0
* g(+∞)=1

In [13]:

**def model(X, theta):**

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



In [14]:

[复制代码](javascript:void(0);)

**pdData.insert(0, 'Ones', 1) # in a try / except structure so as not to return an error if the block si executed several times**

**# set X (training data) and y (target variable)**

**orig\_data = pdData.as\_matrix() # convert the Pandas seful for further computations**

**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)**

**theta = np.zeros([1, 3])**

[复制代码](javascript:void(0);)

In [15]:

**X[:5]**

Out[15]:

**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 [16]:

**y[:5]**

Out[16]:

array([[ 0.],

[ 0.],

[ 0.],

[ 1.],

[ 1.]])

In [17]:

**theta**

Out[17]:

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

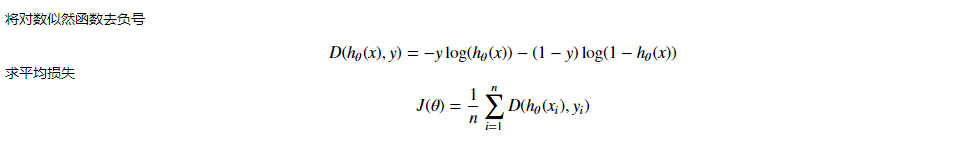
In [18]:

**X.shape, y.shape, theta.shape**

Out[18]:

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

**损失函数**



In [19]:

**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))**

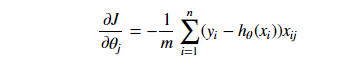
In [20]:

**cost(X, y, theta)**

Out[20]:

**0.69314718055994529**

**计算梯度**



In [21]:

[复制代码](javascript:void(0);)

**def gradient(X, y, theta):**

**grad = np.zeros(theta.shape) # （1,3）**

**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**

[复制代码](javascript:void(0);)

**Gradient descent**

比较3中不同梯度下降方法

In [22]:

[复制代码](javascript:void(0);)

**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**

[复制代码](javascript:void(0);)

In [23]:

[复制代码](javascript:void(0);)

**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**

[复制代码](javascript:void(0);)

In [24]:

[复制代码](javascript:void(0);)

**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**

[复制代码](javascript:void(0);)

In [25]:

[复制代码](javascript:void(0);)

**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**

[复制代码](javascript:void(0);)

**不同的停止策略**

**设定迭代次数**

In [26]:

**#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 - Duration: 1.52s

Out[26]:

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 [27]:**

**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 - Duration: 39.86s

Out[27]:

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 [28]:

**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 - Duration: 13.44s

Out[28]:

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

**对比不同的梯度下降方法**

**Stochastic descent**

In [29]:

**runExpe(orig\_data, theta, 1, STOP\_ITER, thresh=5000, alpha=0.001)**

\*\*\*Original data - learning rate: 0.001 - Stochastic descent - Stop: 5000 iterations

Theta: [[-0.38585397 0.09042018 -0.01044445]] - Iter: 5000 - Last cost: 1.53 - Duration: 0.48s

Out[29]:

array([[-0.38585397, 0.09042018, -0.01044445]])

**有点爆炸。。。很不稳定,再来试试把学习率调小一些**

In [30]:

**runExpe(orig\_data, theta, 1, STOP\_ITER, thresh=15000, alpha=0.000002)**

\*\*\*Original data - learning rate: 2e-06 - Stochastic descent - Stop: 15000 iterations

Theta: [[-0.00201963 0.01014321 0.00107125]] - Iter: 15000 - Last cost: 0.63 - Duration: 1.70s

Out[30]:

array([[-0.00201963, 0.01014321, 0.00107125]])

**速度快，但稳定性差，需要很小的学习率**

**Mini-batch descent**

In [31]:

**runExpe(orig\_data, theta, 16, STOP\_ITER, thresh=15000, alpha=0.001)**

\*\*\*Original data - learning rate: 0.001 - Mini-batch (16) descent - Stop: 15000 iterations

Theta: [[-1.032863 0.03624659 0.02571257]] - Iter: 15000 - Last cost: 0.97 - Duration: 2.11s

Out[31]:

array([[-1.032863 , 0.03624659, 0.02571257]])

**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 [32]:

**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 - Duration: 1.92s

Out[32]:

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 [33]:

**runExpe(scaled\_data, theta, n, STOP\_GRAD, thresh=0.02, alpha=0.001)**

\*\*\*Scaled data - learning rate: 0.001 - Gradient descent - Stop: gradient norm < 0.02

Theta: [[ 1.0707921 2.63030842 2.41079787]] - Iter: 59422 - Last cost: 0.22 - Duration: 21.58s

Out[33]:

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 [34]:

**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: [[ 1.14904527 2.7920262 2.56725991]] - Iter: 72624 - Last cost: 0.22 - Duration: 9.14s

**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 [35]:

**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 norm < 0.004

Theta: [[ 1.16033549 2.81496841 2.59589695]] - Iter: 2393 - Last cost: 0.22 - Duration: 0.40s

Out[35]:

array([[ 1.16033549, 2.81496841, 2.59589695]])

**精度**

In [36]:

**#设定阈值**

**def predict(X, theta):**

**return [1 if x >= 0.5 else 0 for x in model(X, theta)]**

In [37]:

**scaled\_X = scaled\_data[:, :3]**

**y = scaled\_data[:, 3]**

**predictions = predict(scaled\_X, theta)**

**correct = [1 if ((a == 1 and b == 1) or (a == 0 and b == 0)) else 0 for (a, b) in zip(predictions, y)]**

**accuracy = (sum(map(int, correct)) % len(correct))**

**print ('accuracy = {0}%'.format(accuracy))**

**accuracy = 89%**