

# 详解 | 如何用Python实现机器学习算法

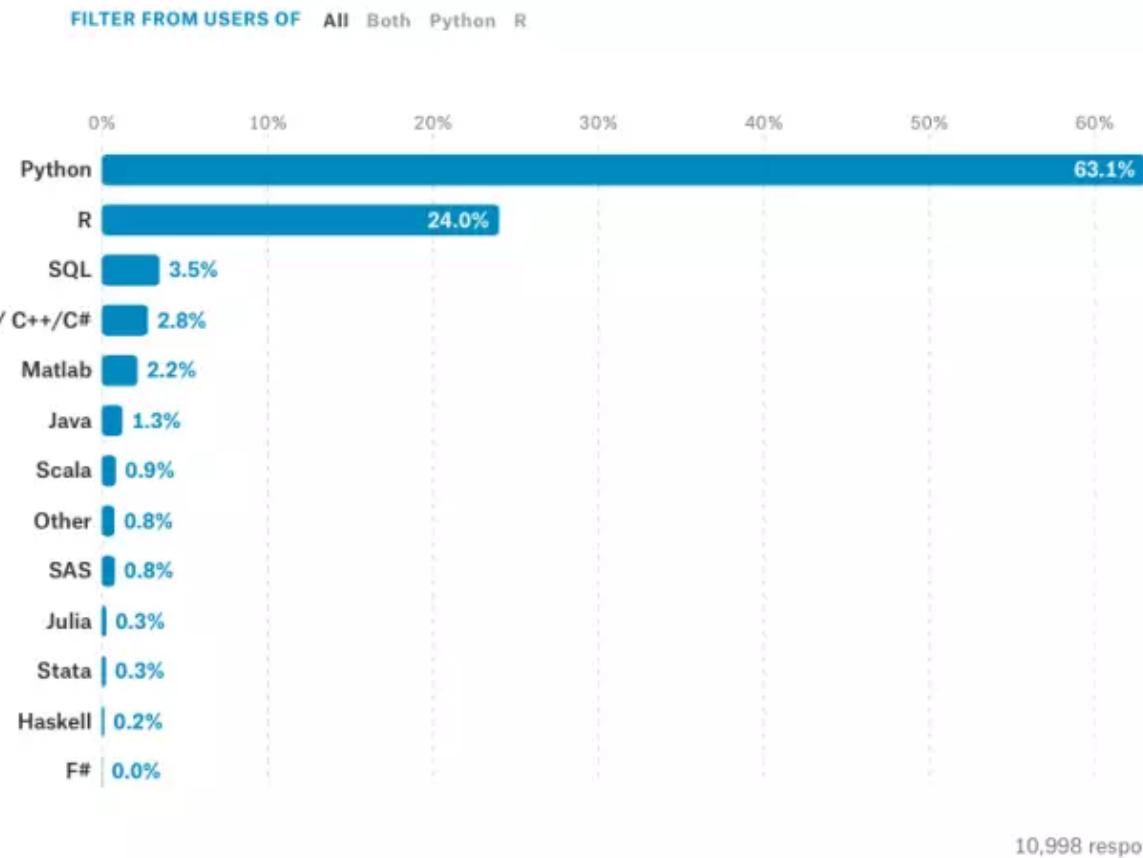
2017-11-21 Lawlite AI科技大本营



作者 | Lawlite

人生苦短，就用 Python。

在 Kaggle 最新发布的全球数据科学/机器学习现状报告中，来自 50 多个国家的 16000 多位从业者纷纷向新手们推荐 Python 语言，用以学习机器学习。



那么，用Python实现出来的机器学习算法都是什么样子呢？营长刚好在GitHub上发现了东南大学研究生“Lawlite”的一个项目——机器学习算法的Python实现，下面从线性回归到反向传播算法、从SVM到K-means聚类算法，咱们一一来分析其中的Python代码。

## 目录

- **一、线性回归**
  - 1、代价函数
  - 2、梯度下降算法
  - 3、均值归一化
  - 4、最终运行结果
  - 5、使用scikit-learn库中的线性模型实现
- **二、逻辑回归**
  - 1、代价函数
  - 2、梯度
  - 3、正则化
  - 4、S型函数（即）
  - 5、映射为多项式
  - 6、使用的优化方法
  - 7、运行结果

- 8、使用scikit-learn库中的逻辑回归模型实现

- 逻辑回归\_手写数字识别\_OneVsAll

- 1、随机显示100个数字
- 2、OneVsAll
- 3、手写数字识别
- 4、预测
- 5、运行结果
- 6、使用scikit-learn库中的逻辑回归模型实现

- **三、BP神经网络**

- 1、神经网络model
- 2、代价函数
- 3、正则化
- 4、反向传播BP
- 5、BP可以求梯度的原因
- 6、梯度检查
- 7、权重的随机初始化
- 8、预测
- 9、输出结果

- **四、SVM支持向量机**

- 1、代价函数
- 2、Large Margin
- 3、SVM Kernel ( 核函数 )
- 4、使用中的模型代码
- 5、运行结果

- **五、K-Means聚类算法**

- 1、聚类过程
- 2、目标函数
- 3、聚类中心的选择
- 4、聚类个数K的选择
- 5、应用——图片压缩
- 6、使用scikit-learn库中的线性模型实现聚类
- 7、运行结果

- **六、PCA主成分分析(降维)**

- 1、用处
- 2、 $2D \rightarrow 1D$ ,  $nD \rightarrow kD$
- 3、主成分分析PCA与线性回归的区别
- 4、PCA降维过程

- 5、数据恢复
- 6、主成分个数的选择（即要降的维度）
- 7、使用建议
- 8、运行结果
- 9、使用scikit-learn库中的PCA实现降维

## ■ 七、异常检测 Anomaly Detection

- 1、高斯分布（正态分布）
- 2、异常检测算法
- 3、评价的好坏，以及的选取
- 4、选择使用什么样的feature（单元高斯分布）
- 5、多元高斯分布
- 6、单元和多元高斯分布特点
- 7、程序运行结果

## 正文

### 一、线性回归

[https://github.com/lawlite19/MachineLearning\\_Python/tree/master/LinearRegression](https://github.com/lawlite19/MachineLearning_Python/tree/master/LinearRegression)

全部代码

[https://github.com/lawlite19/MachineLearning\\_Python/blob/master/LinearRegression/LinearRegression.py](https://github.com/lawlite19/MachineLearning_Python/blob/master/LinearRegression/LinearRegression.py)

#### 1、代价函数

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

其中： $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots$

下面就是要求出theta，使代价最小，即代表我们拟合出来的方程距离真实值最近

共有m条数据，其中 $(h_\theta(x^{(i)}) - y^{(i)})^2$ 代表我们要拟合出来的方程到真实值距离的平方，平方的原因是因为可能有负值，正负可能会抵消

前面有系数2的原因是下面求梯度是对每个变量求偏导，2可以消去

实现代码：

```
# 计算代价函数
def computerCost(X,y,theta):
    m = len(y)
    J = 0

    J = (np.transpose(X*theta-y))*(X*theta-y)/(2*m) #计算代价J
    return J
```

注意这里的X是真实数据前加了一列1，因为有theta(0)

## 2、梯度下降算法

代价函数对 $\theta_j$ 求偏导得到：
$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^m [(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}]$$

所以对theta的更新可以写为：

$$\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m [(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}]$$

其中 $\alpha$ 为学习速率，控制梯度下降的速度，一般取0.01, 0.03, 0.1, 0.3.....

为什么梯度下降可以逐步减小代价函数？

假设函数 $f(x)$

泰勒展开： $f(x + \Delta x) = f(x) + f'(x) * \Delta x + o(\Delta x)$ ，

令： $\Delta x = -\alpha * f'(x)$ ，即负梯度方向乘以一个很小的步长 $\alpha$

将 $\Delta x$ 代入泰勒展开式中： $f(x + x) = f(x) - \alpha * [f'(x)]^2 + o(\Delta x)$

可以看出， $\alpha$ 是取得很小的正数， $[f'(x)]^2$ 也是正数，所以可以得出： $f(x + \Delta x) \leq f(x)$

所以沿着负梯度放下，函数在减小，多维情况一样。

```
# 梯度下降算法
def gradientDescent(X,y,theta,alpha,num_iters):
    m = len(y)
    n = len(theta)

    temp = np.matrix(np.zeros((n,num_iters))) # 暂存每次迭代计算的theta，转化为矩阵形式

    J_history = np.zeros((num_iters,1)) #记录每次迭代计算的代价值

    for i in range(num_iters): # 遍历迭代次数
        h = np.dot(X,theta) # 计算内积，matrix可以直接乘
        temp[:,i] = theta - ((alpha/m)*(np.dot(np.transpose(X),h-y))) #梯度的计算
        theta = temp[:,i]
        J_history[i] = computerCost(X,y,theta) #调用计算代价函数
        print '.',
    return theta,J_history
```

### 3、均值归一化

目的是使数据都缩放到一个范围内，便于使用梯度下降算法

其中  $\bar{x}$  为所有此feature数据的平均值

可以是最大值-最小值，也可以是这个feature对应的数据的标准差

实现代码：

```
# 归一化feature
def featureNormaliza(X):
    X_norm = np.array(X) #将X转化为numpy数组对象，才可以进行矩阵的运算
    #定义所需变量
    mu = np.zeros((1,X.shape[1]))
    sigma = np.zeros((1,X.shape[1]))

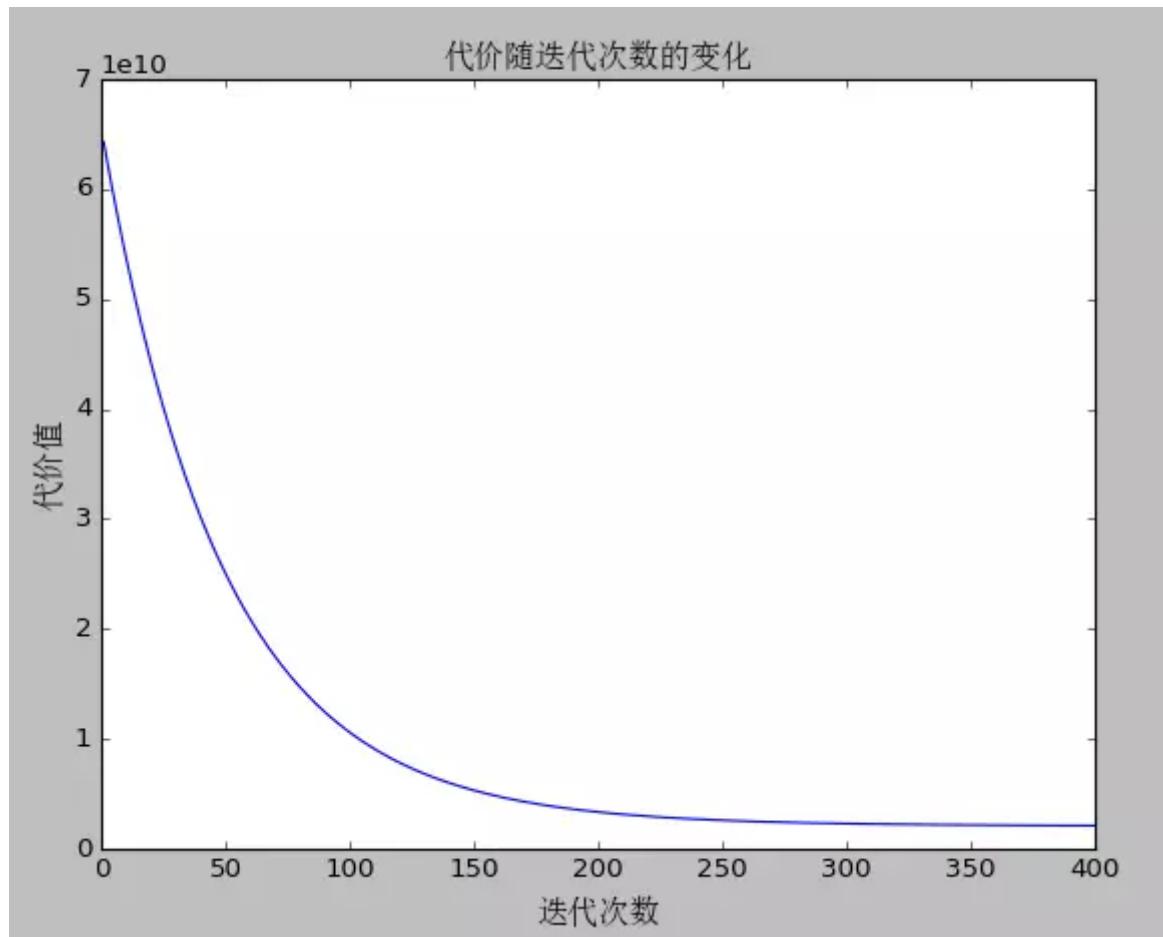
    mu = np.mean(X_norm,0) # 求每一列的平均值 ( 0指定为列，1代表行 )
    sigma = np.std(X_norm,0) # 求每一列的标准差
    for i in range(X.shape[1]): # 遍历列
        X_norm[:,i] = (X_norm[:,i]-mu[i])/sigma[i] # 归一化
```

```
return X_norm,mu,sigma
```

注意预测的时候也需要均值归一化数据

#### 4、最终运行结果

代价随迭代次数的变化



#### 5、使用scikit-learn库中的线性模型实现

[https://github.com/lawlite19/MachineLearning\\_Python/blob/master/LinearRegression/LinearRegression\\_scikit-learn.py](https://github.com/lawlite19/MachineLearning_Python/blob/master/LinearRegression/LinearRegression_scikit-learn.py)

导入包

```
from sklearn import linear_model  
from sklearn.preprocessing import StandardScaler #引入缩放的包
```

归一化

```
# 归一化操作
```

```

scaler = StandardScaler()
scaler.fit(X)
x_train = scaler.transform(X)
x_test = scaler.transform(np.array([1650,3]))

```

### 线性模型拟合

```

# 线性模型拟合
model = linear_model.LinearRegression()
model.fit(x_train, y)

```

### 预测

```

#预测结果
result = model.predict(x_test)

```

## 二、逻辑回归

[https://github.com/lawlite19/MachineLearning\\_Python/tree/master/LogisticRegression](https://github.com/lawlite19/MachineLearning_Python/tree/master/LogisticRegression)

### 全部代码

[https://github.com/lawlite19/MachineLearning\\_Python/blob/master/LogisticRegression/LogisticRegression.py](https://github.com/lawlite19/MachineLearning_Python/blob/master/LogisticRegression/LogisticRegression.py)

### 1、代价函数

$$\begin{cases} J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{cost}(h_\theta(x^{(i)}), y^{(i)}) \\ \text{cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & y = 1 \\ -\log(1 - h_\theta(x)) & y = 0 \end{cases} \end{cases}$$

可以综合起来为：

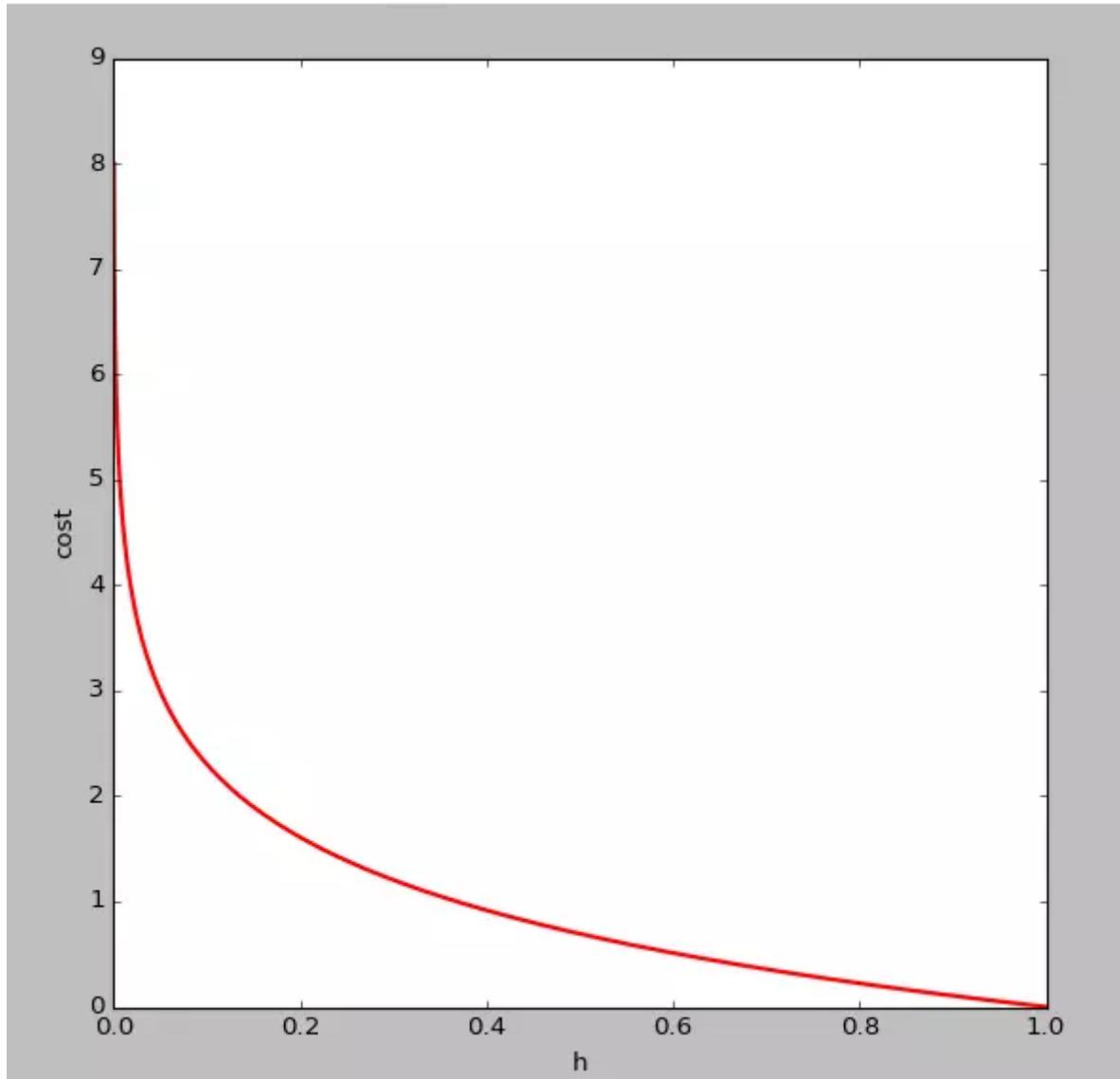
$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))]$$

其中：

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

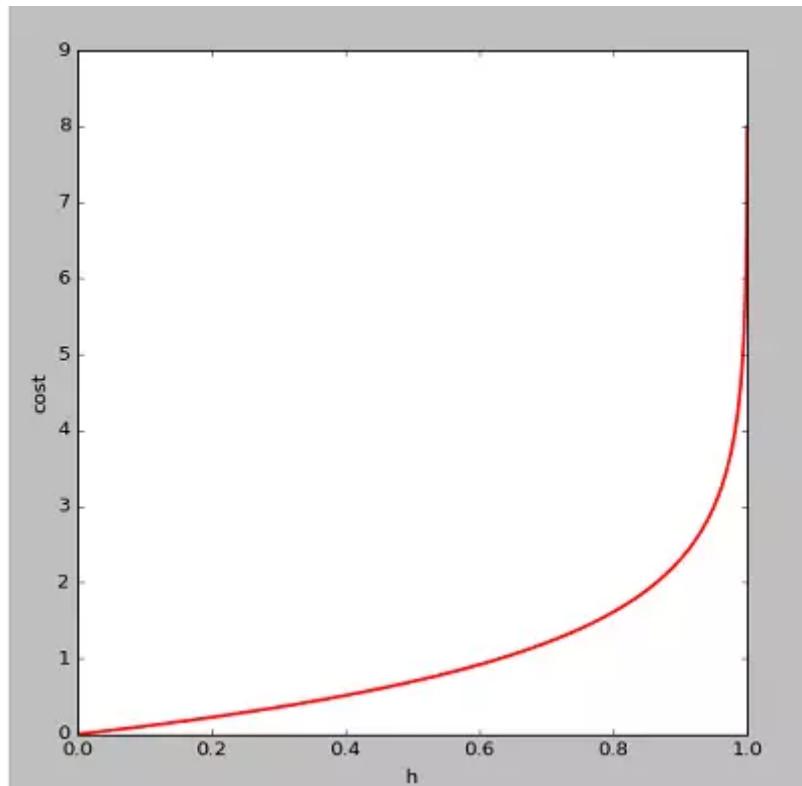
为什么不用线性回归的代价函数表示，因为线性回归的代价函数可能是非凸的，对于分类问题，使用梯度下降很难得到最小值，上面的代价函数是凸函数

$-\log(h_{\theta}(x))$  的图像如下，即  $y=1$  时：



可以看出，当  $h_{\theta}(x)$  趋于 1， $y=1$ ，与预测值一致，此时付出的代价 cost 趋于 0，若  $h_{\theta}(x)$  趋于 0， $y=1$ ，此时的代价 cost 值非常大，我们最终的目的是最小化代价值

同理  $-\log(1-h_{\theta}(x))$  的图像如下 ( $y=0$ )：



## 2、梯度

同样对代价函数求偏导：

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^m [(h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}]$$

可以看出与线性回归的偏导数一致

推导过程

推导:  $\frac{\partial J(\theta)}{\partial \theta_j} = -\frac{1}{m} \sum_{i=1}^m \left( y^{(i)} \cdot \frac{1}{g(\theta^T x^{(i)})} \cdot \frac{\partial h_\theta(x^{(i)})}{\partial \theta_j} + (1-y^{(i)}) \cdot \frac{1}{1-g(\theta^T x^{(i)})} \left( -\frac{\partial h_\theta(x^{(i)})}{\partial \theta_j} \right) \right)$

令  $g(z) = \frac{1}{1+e^{-z}}$ , 则  $h_\theta(x) = g(\theta^T x)$

$$= -\frac{1}{m} \sum_{i=1}^m \left( y^{(i)} \frac{1}{g(\theta^T x^{(i)})} - (1-y^{(i)}) \frac{1}{1-g(\theta^T x^{(i)})} \right) \frac{\partial g(\theta^T x^{(i)})}{\partial \theta_j}$$

$$\left( \frac{1}{1+e^{-x}} \right)' = [(1+e^{-x})^{-1}]' = -1 \cdot (1+e^{-x})^{-2} \cdot e^{-x} \cdot (-1) = \frac{e^{-x}}{(1+e^{-x})^2}$$

$$= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} = \frac{1}{1+e^{-x}} \left( 1 - \frac{1}{1+e^{-x}} \right)$$

$\therefore$  原式  $= -\frac{1}{m} \sum_{i=1}^m \left( y^{(i)} \frac{1}{g(\theta^T x^{(i)})} - (1-y^{(i)}) \frac{1}{1-g(\theta^T x^{(i)})} \right) \cdot \underbrace{g(\theta^T x^{(i)}) (1-g(\theta^T x^{(i)}))}_{\text{括号内}} \cdot \frac{\partial \theta^T x^{(i)}}{\partial \theta_j}$

$$= -\frac{1}{m} \sum_{i=1}^m \left[ y^{(i)} (1-g(\theta^T x^{(i)})) - (1-y^{(i)}) g(\theta^T x^{(i)}) \right] \cdot x_j^{(i)}$$

$$= \frac{1}{m} \sum_{i=1}^m (g(\theta^T x^{(i)}) - y^{(i)}) x_j^{(i)}$$

### 3. 正则化

目的是为了防止过拟合

在代价函数中加上一项

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_\theta(x^{(i)}) + (1-y^{(i)}) \log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

注意j是重1开始的，因为theta(0)为一个常数项，X中最前面一列会加上1列1，所以乘积还是theta(0), feature没有关系，没有必要正则化

正则化后的代价：

```
# 代价函数
def costFunction(initial_theta,X,y,inital_lambda):
    m = len(y)
    J = 0
```

```

h = sigmoid(np.dot(X,initial_theta)) # 计算h(z)
theta1 = initial_theta.copy()       # 因为正则化j=1从1开始，不包含0，所以复制一份，前theta(0)值为
0
theta1[0] = 0

temp = np.dot(np.transpose(theta1),theta1)
J = (-np.dot(np.transpose(y),np.log(h))-np.dot(np.transpose(1-y),np.log(1-
h))+temp*inital_lambda/2)/m # 正则化的代价方程
return J

```

正则化后的代价的梯度

```

# 计算梯度

def gradient(initial_theta,X,y,inital_lambda):
    m = len(y)
    grad = np.zeros((initial_theta.shape[0]))

    h = sigmoid(np.dot(X,initial_theta))# 计算h(z)
    theta1 = initial_theta.copy()
    theta1[0] = 0

    grad = np.dot(np.transpose(X),h-y)/m+inital_lambda/m*theta1 #正则化的梯度
    return grad

```

#### 4、S型函数（即 ）

实现代码：

```

# S型函数
def sigmoid(z):
    h = np.zeros((len(z),1)) # 初始化，与z的长度一致

    h = 1.0/(1.0+np.exp(-z)) return h

```

#### 5、映射为多项式

因为数据的feature可能很少，导致偏差大，所以创造出一些feature结合

eg:映射为2次方的形式:

实现代码：

```
# 映射为多项式
def mapFeature(X1,X2):
    degree = 3;           # 映射的最高次方
    out = np.ones((X1.shape[0],1)) # 映射后的结果数组 ( 取代X )
    ...
    这里以degree=2为例，映射为1,x1,x2,x1^2,x1,x2,x2^2
    ...
    for i in np.arange(1,degree+1):
        for j in range(i+1):
            temp = X1***(i-j)*(X2**j) # 矩阵直接乘相当于matlab中的点乘.*
            out = np.hstack((out, temp.reshape(-1,1)))
    return out
```

## 6、使用scipy的优化方法

梯度下降使用scipy中optimize中的fmin\_bfgs函数

调用scipy中的优化算法fmin\_bfgs ( 拟牛顿法Broyden-Fletcher-Goldfarb-Shanno  
costFunction是自己实现的一个求代价的函数 ,

initial\_theta表示初始化的值,

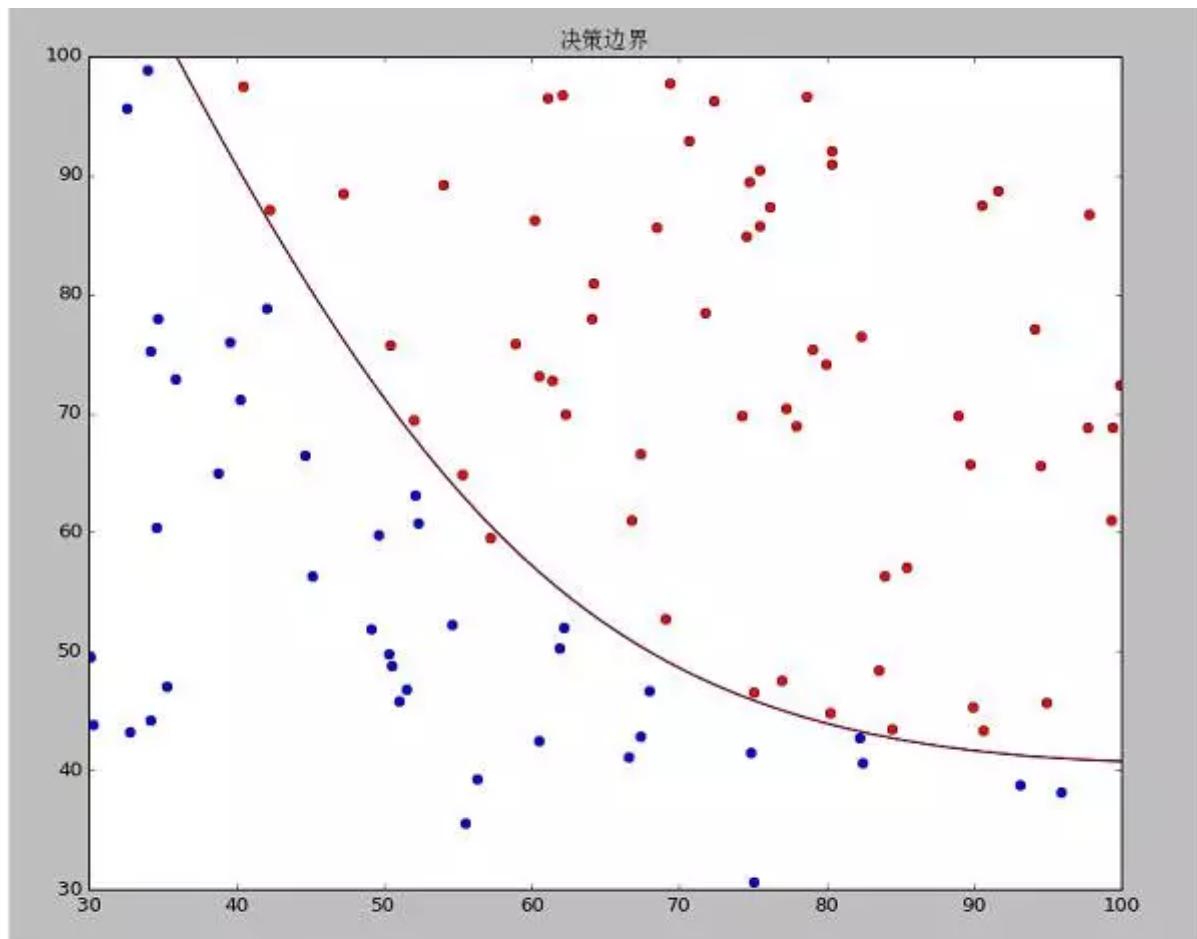
fprime指定costFunction的梯度

args是其余参数 , 以元组的形式传入 , 最后会将最小化costFunction的theta返回

```
result = optimize.fmin_bfgs(costFunction, initial_theta, fprime=gradient, args=(X,y,initial_lambda))
```

## 7、运行结果

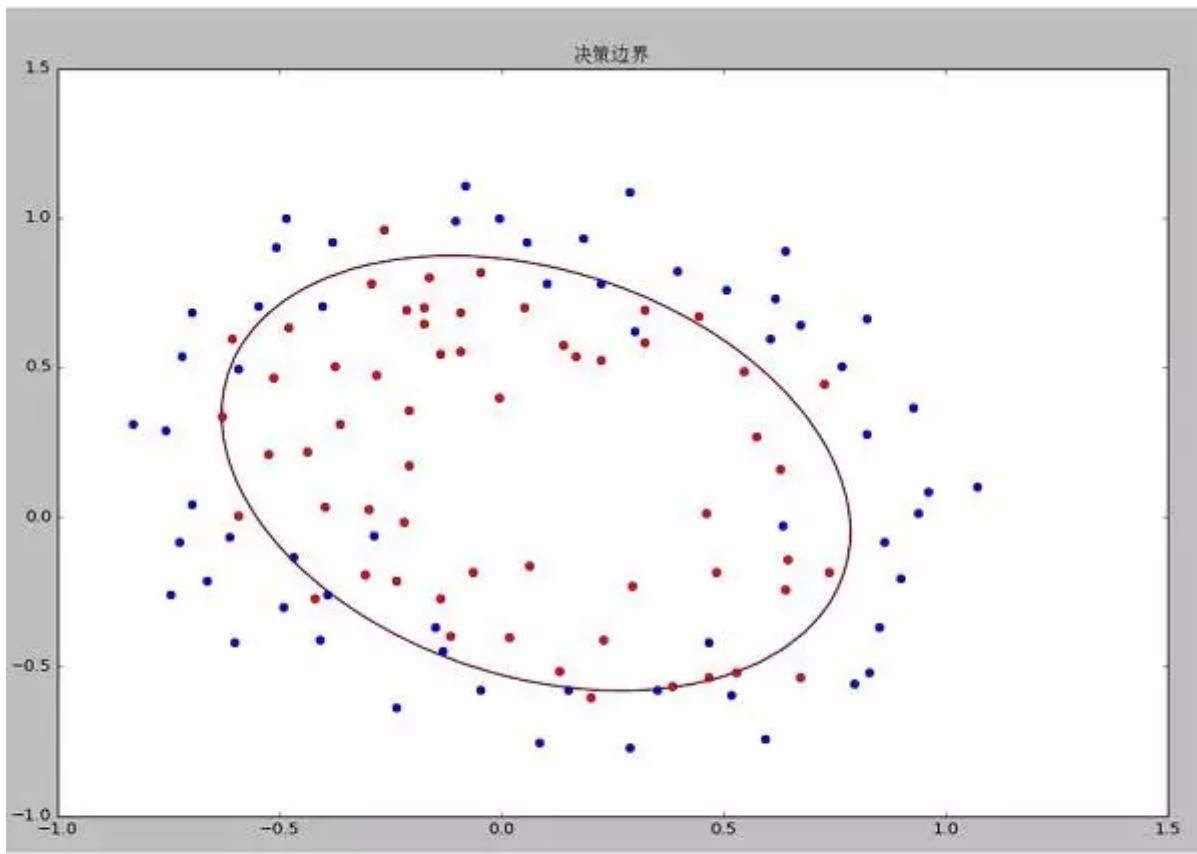
data1决策边界和准确度



Iterations: 11  
Function evaluations: 40  
Gradient evaluations: 40

在训练集上的准确度为99.000000%

data2决策边界和准确度



```
Optimization terminated successfully.
    Current function value: 0.432431
    Iterations: 37
    Function evaluations: 39
    Gradient evaluations: 39
```

在训练集上的准确度为83.050847%

## 8、使用scikit-learn库中的逻辑回归模型实现

[https://github.com/lawlite19/MachineLearning\\_Python/blob/master/LogisticRegression/LogisticRegression\\_scikit-learn.py](https://github.com/lawlite19/MachineLearning_Python/blob/master/LogisticRegression/LogisticRegression_scikit-learn.py)

### 导入包

```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.cross_validation import train_test_split
import numpy as np
```

### 划分训练集和测试集

```
# 划分为训练集和测试集
x_train,x_test,y_train,y_test = train_test_split(X,y,test_size=0.2)
```

### 归一化

```
# 归一化
scaler = StandardScaler()
```

```
scaler.fit(x_train)
x_train = scaler.fit_transform(x_train)
x_test = scaler.fit_transform(x_test)
```

## 逻辑回归

```
#逻辑回归
model = LogisticRegression()
model.fit(x_train,y_train)
```

## 预测

```
# 预测
predict = model.predict(x_test)
right = sum(predict == y_test)
predict = np.hstack((predict.reshape(-1,1),y_test.reshape(-1,1))) # 将预测值和真实值放在一块，好观察
print predict
print ('测试集准确率 : %f%%' %(right*100.0/predict.shape[0])) #计算在测试集上的准确度
```

## 逻辑回归\_手写数字识别\_OneVsAll

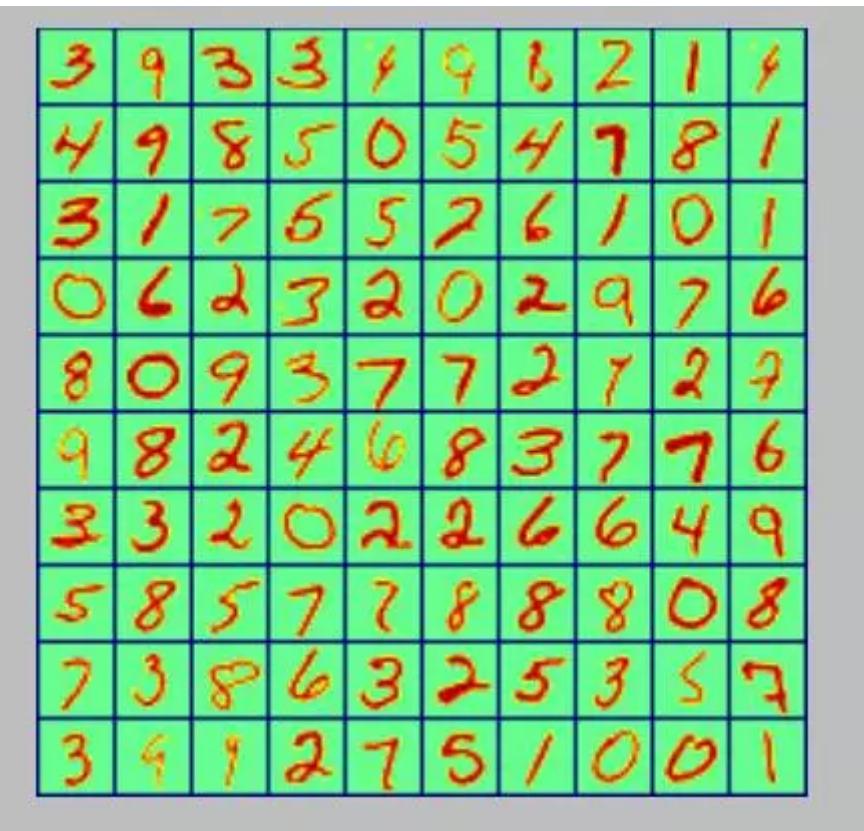
[https://github.com/lawlite19/MachineLearning\\_Python/blob/master/LogisticRegression](https://github.com/lawlite19/MachineLearning_Python/blob/master/LogisticRegression)

## 全部代码

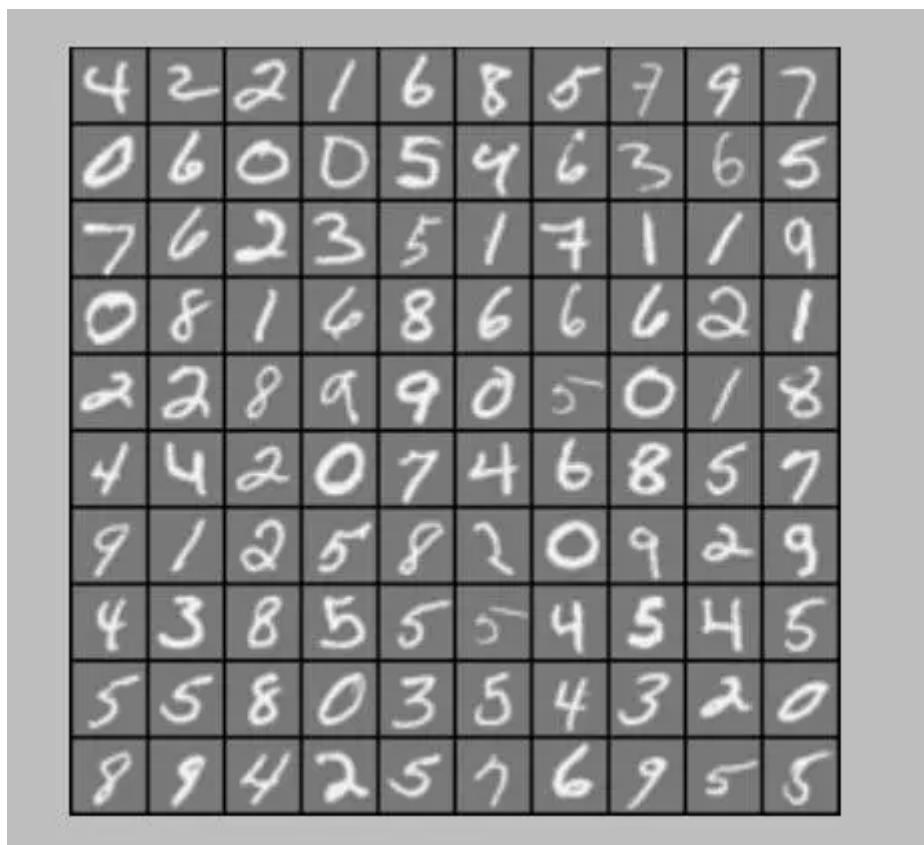
[https://github.com/lawlite19/MachineLearning\\_Python/blob/master/LogisticRegression/LogisticRegression\\_OneVsAll.py](https://github.com/lawlite19/MachineLearning_Python/blob/master/LogisticRegression/LogisticRegression_OneVsAll.py)

## 1、随机显示100个数字

我没有使用scikit-learn中的数据集，像素是20\*20px，彩色图如下



灰度图：



实现代码：

```
# 显示100个数字
```

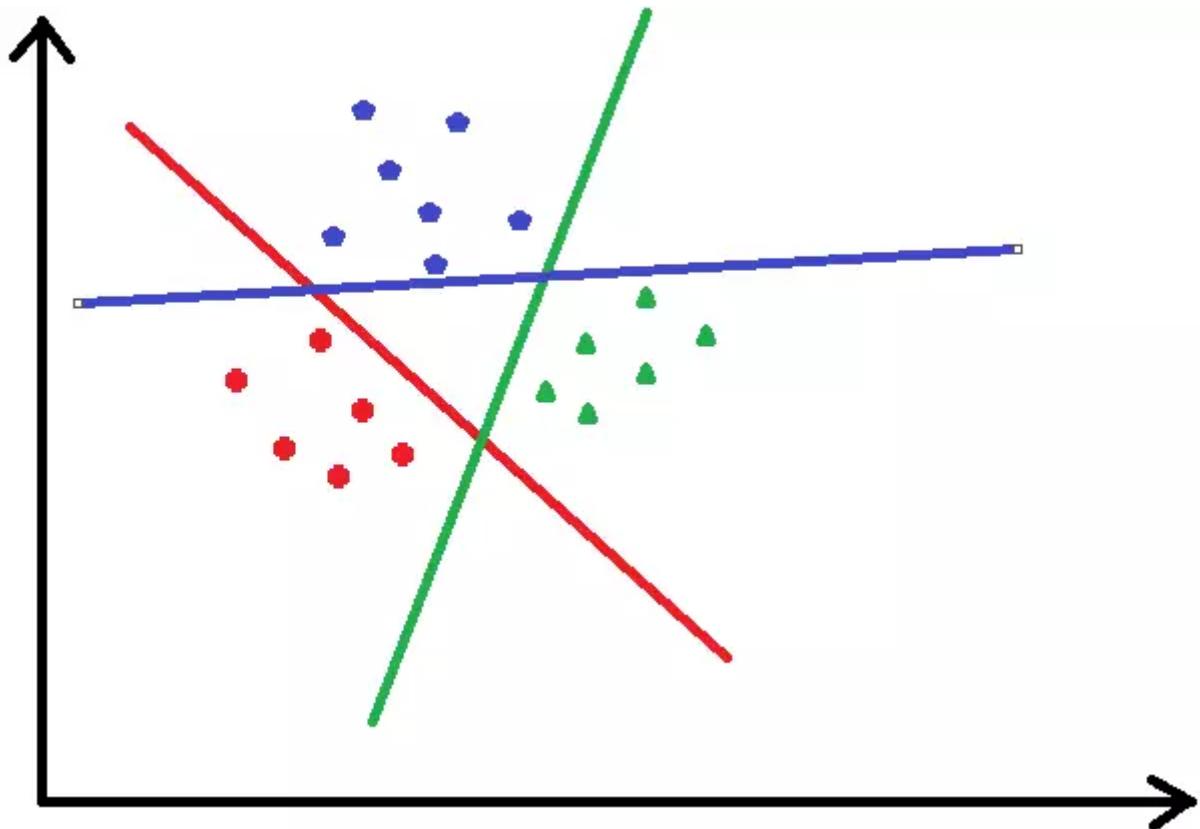
```
def display_data(imgData):
    sum = 0
    ...
    显示100个数（若是一个一个绘制将会非常慢，可以将要画的数字整理好，放到一个矩阵中，显示这个矩阵即可）
    - 初始化一个二维数组
    - 将每行的数据调整成图像的矩阵，放进二维数组
    - 显示即可
    ...
    pad = 1
    display_array = -np.ones((pad+10*(20+pad),pad+10*(20+pad)))
    for i in range(10):
        for j in range(10):
            display_array[pad+i*(20+pad):pad+i*(20+pad)+20,pad+j*(20+pad):pad+j*(20+pad)+20] =
            (imgData[sum,:].reshape(20,20,order="F")) # order=F指定以列优先，在matlab中是这样的，python中
            需要指定，默认以行
            sum += 1

    plt.imshow(display_array,cmap='gray') #显示灰度图像
    plt.axis('off')
    plt.show()
```

## 2、OneVsAll

如何利用逻辑回归解决多分类的问题，OneVsAll就是把当前某一类看成一类，其他所有类别看作一类，这样就成了二分类的问题了

如下图，把途中的数据分成三类，先把红色的看成一类，把其他的看作另外一类，进行逻辑回归，然后把蓝色的看成一类，其他的再看成一类，以此类推...



可以看出大于2类的情况下，有多少类就要进行多少次的逻辑回归分类

### 3、手写数字识别

共有0-9，10个数字，需要10次分类

由于数据集y给出的是0,1,2...9的数字，而进行逻辑回归需要0/1的label标记，所以需要对y处理

说一下数据集，前500个是0,500-1000是1,...,所以如下图，处理后的y，**前500行的第一列是1，其余都是0,500-1000行第二列是1，其余都是0....**

	A	B	C	D	E	F	G	H	I	J
490	1	0	0	0	0	0	0	0	0	0
491	1	0	0	0	0	0	0	0	0	0
492	1	0	0	0	0	0	0	0	0	0
493	1	0	0	0	0	0	0	0	0	0
494	1	0	0	0	0	0	0	0	0	0
495	1	0	0	0	0	0	0	0	0	0
496	1	0	0	0	0	0	0	0	0	0
497	1	0	0	0	0	0	0	0	0	0
498	1	0	0	0	0	0	0	0	0	0
499	1	0	0	0	0	0	0	0	0	0
500	1	0	0	0	0	0	0	0	0	0
501	0	1	0	0	0	0	0	0	0	0
502	0	1	0	0	0	0	0	0	0	0
503	0	1	0	0	0	0	0	0	0	0
504	0	1	0	0	0	0	0	0	0	0
505	0	1	0	0	0	0	0	0	0	0
506	0	1	0	0	0	0	0	0	0	0
507	0	1	0	0	0	0	0	0	0	0
508	0	1	0	0	0	0	0	0	0	0
509	0	1	0	0	0	0	0	0	0	0
510	0	1	0	0	0	0	0	0	0	0
511	0	1	0	0	0	0	0	0	0	0
512	0	1	0	0	0	0	0	0	0	0
513	0	1	0	0	0	0	0	0	0	0
514	0	1	0	0	0	0	0	0	0	0
515	0	1	0	0	0	0	0	0	0	0

然后调用梯度下降算法求解theta

实现代码：

```
# 求每个分类的theta，最后返回所有的all_theta
def oneVsAll(X,y,num_labels,Lambda):
    # 初始化变量
    m,n = X.shape
    all_theta = np.zeros((n+1,num_labels)) # 每一列对应相应分类的theta,共10列
    X = np.hstack((np.ones((m,1)),X))      # X前补上一列1的偏置bias
    class_y = np.zeros((m,num_labels))      # 数据的y对应0-9，需要映射为0/1的关系
    initial_theta = np.zeros((n+1,1))       # 初始化一个分类的theta

    # 映射y
    for i in range(num_labels):
        class_y[:,i] = np.int32(y==i).reshape(1,-1) # 注意reshape(1,-1)才可以赋值

    #np.savetxt("class_y.csv", class_y[0:600,:], delimiter=',')

    """遍历每个分类，计算对应的theta值"""
    for i in range(num_labels):
        result = optimize.fmin_bfgs(costFunction, initial_theta, fprime=gradient, args=(X,class_y[:,i],Lambda)) # 调用梯度下降的优化方法
        all_theta[:,i] = result.reshape(1,-1) # 放入all_theta中
```

```
all_theta = np.transpose(all_theta)
return all_theta
```

## 4、预测

之前说过，预测的结果是一个概率值，利用学习出来的theta代入预测的S型函数中，每行的最大值就是某个数字的最大概率，所在的列号就是预测的数字的真实值，因为在分类时，所有为0的将y映射在第一列，为1的映射在第二列，依次类推

实现代码：

```
# 预测
def predict_oneVsAll(all_theta,X):
    m = X.shape[0]
    num_labels = all_theta.shape[0]
    p = np.zeros((m,1))
    X = np.hstack((np.ones((m,1)),X)) #在X最前面加一列1

    h = sigmoid(np.dot(X,np.transpose(all_theta))) #预测
```

...

返回h中每一行最大值所在的列号

- np.max(h, axis=1)返回h中每一行的最大值（是某个数字的最大概率）

- 最后where找到的最大概率所在的列号（列号即是对应的数字）

...

```
p = np.array(np.where(h[0,:] == np.max(h, axis=1)[0]))
for i in np.arange(1, m):
    t = np.array(np.where(h[i,:] == np.max(h, axis=1)[i]))
    p = np.vstack((p,t))
return p
```

## 5、运行结果

10次分类，在训练集上的准确度：

```

Optimization terminated successfully.
    Current function value: 0.018265
    Iterations: 361
    Function evaluations: 363
    Gradient evaluations: 363
Optimization terminated successfully.
    Current function value: 0.030653
    Iterations: 361
    Function evaluations: 364
    Gradient evaluations: 364
Optimization terminated successfully.
    Current function value: 0.078457
    Iterations: 454
    Function evaluations: 456
    Gradient evaluations: 456
Optimization terminated successfully.
    Current function value: 0.071193
    Iterations: 439
    Function evaluations: 441
    Gradient evaluations: 441

```

预测准确度为: 96.480000%

## 6、使用scikit-learn库中的逻辑回归模型实现

[https://github.com/lawlite19/MachineLearning\\_Python/blob/master/LogisticRegression/LogisticRegression\\_OneVsAll\\_scikit-learn.py](https://github.com/lawlite19/MachineLearning_Python/blob/master/LogisticRegression/LogisticRegression_OneVsAll_scikit-learn.py)

### 1、导入包

```

from scipy import io as spio
import numpy as np
from sklearn import svm
from sklearn.linear_model import LogisticRegression

```

### 2、加载数据

```

data = loadmat_data("data_digits.mat")
X = data['X'] # 获取X数据，每一行对应一个数字20x20px
y = data['y'] # 这里读取mat文件y的shape=(5000, 1)
y = np.ravel(y) # 调用sklearn需要转化成一维的(5000,)

```

### 3、拟合模型

```

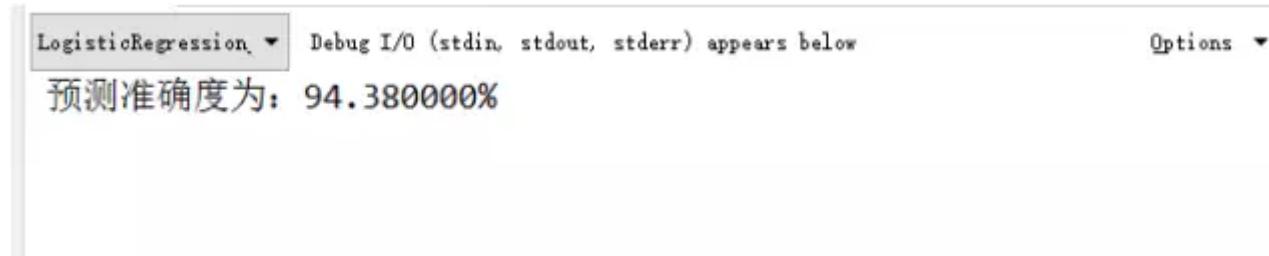
model = LogisticRegression()
model.fit(X, y) # 拟合

```

#### 4、预测

```
predict = model.predict(X) #预测  
print u"预测准确度为 : %f%%"%np.mean(np.float64(predict == y)*100)
```

#### 5、输出结果（在训练集上的准确度）



### 三、BP神经网络

#### 全部代码

[https://github.com/lawlite19/MachineLearning\\_Python/blob/master/NeuralNetwok/NeuralNetwork.py](https://github.com/lawlite19/MachineLearning_Python/blob/master/NeuralNetwok/NeuralNetwork.py)

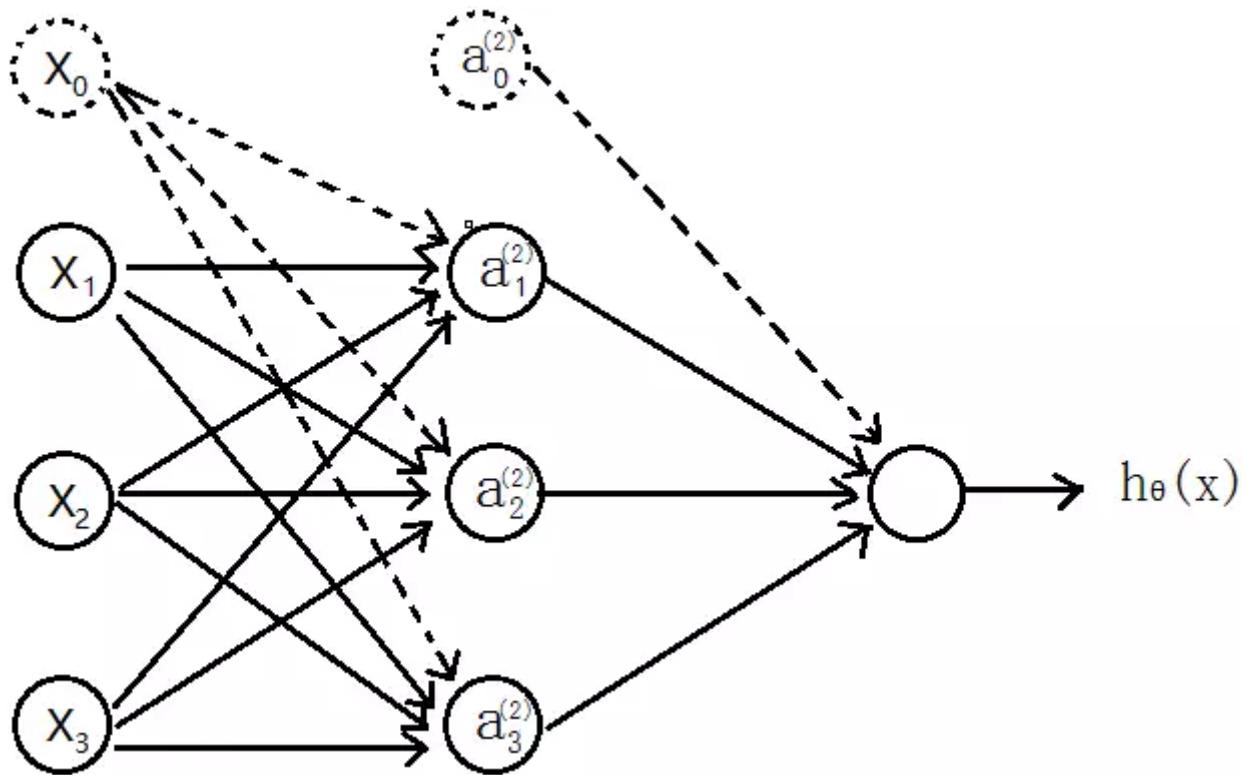
#### 1、神经网络model

先介绍个三层的神经网络，如下图所示

输入层 ( input layer ) 有三个units ( 为补上的bias，通常设为1 )

表示第j层的第i个激励，也称为为单元unit

为第j层到第j+1层映射的权重矩阵，就是每条边的权重



所以可以得到：

隐含层：

$$a_1^{(2)} = g(\theta_{10}^{(1)}x_0 + \theta_{11}^{(1)}x_1 + \theta_{12}^{(1)}x_2 + \theta_{13}^{(1)}x_3)$$

$$a_2^{(2)} = g(\theta_{20}^{(1)}x_0 + \theta_{21}^{(1)}x_1 + \theta_{22}^{(1)}x_2 + \theta_{23}^{(1)}x_3)$$

$$a_3^{(2)} = g(\theta_{30}^{(1)}x_0 + \theta_{31}^{(1)}x_1 + \theta_{32}^{(1)}x_2 + \theta_{33}^{(1)}x_3)$$

$$\text{输出层 } h_\theta(x) = a_1^{(3)} = g(\theta_{10}^{(2)}a_0^{(2)} + \theta_{11}^{(2)}a_1^{(2)} + \theta_{12}^{(2)}a_2^{(2)} + \theta_{13}^{(2)}a_3^{(2)}) ,$$

其中，S型函数  $g(z) = \frac{1}{1+e^{-z}}$ ，也成为激励函数

可以看出  $\theta^{(1)}$  为  $3 \times 4$  的矩阵， $\theta^{(2)}$  为  $1 \times 4$  的矩阵

$\Rightarrow j+1$  的单元数  $\times (j$  层的单元数 + 1 )

## 2、代价函数

假设最后输出的 $(h_{\Theta}(x))_i$ ，即代表输出层有K个单元

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K [y_k^{(i)} \log(h_{\Theta}(x^{(i)}))_k + (1-y_k^{(i)}) \log(1-h_{\Theta}(x^{(i)}))_k]$$

其中， $(h_{\Theta}(x))_i$ 代表第i个单元输出与逻辑回归的代价函数

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_{\theta}(x^{(i)})) + (1-y^{(i)}) \log(1-h_{\theta}(x^{(i)}))]$$

差不多，就是累加上每个输出（共有K个输出）

### 3、正则化

L-->所有层的个数

$S_l$ -->第l层unit的个数

正则化后的代价函数为

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K [y_k^{(i)} \log(h_{\Theta}(x^{(i)}))_k + (1-y_k^{(i)}) \log(1-h_{\Theta}(x^{(i)}))_k] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{S_l} \sum_{j=1}^{S_{l+1}} (\theta_{ij}^{(l)})^2$$

$\theta$ 共有L-1层，然后是累加对应每一层的theta矩阵，注意不包含加上偏置项对应的theta(0)

正则化后的代价函数实现代码：

```
# 代价函数
def nnCostFunction(nn_params,input_layer_size,hidden_layer_size,num_labels,X,y,Lambda):
    length = nn_params.shape[0] # theta的中长度
    # 还原theta1和theta2
    Theta1 = nn_params[0:hidden_layer_size*(input_layer_size+1)].reshape(hidden_layer_size,input_layer_size+1)
    Theta2 = nn_params[hidden_layer_size*(input_layer_size+1):length].reshape(num_labels,hidden_layer_size+1)

    # np.savetxt("Theta1.csv",Theta1,delimiter=',')
```

```

m = X.shape[0]
class_y = np.zeros((m,num_labels))    # 数据的y对应0-9，需要映射为0/1的关系
# 映射y
for i in range(num_labels):
    class_y[:,i] = np.int32(y==i).reshape(1,-1) # 注意reshape(1,-1)才可以赋值

'''去掉theta1和theta2的第一列，因为正则化时从1开始'''
Theta1_colCount = Theta1.shape[1]
Theta1_x = Theta1[:,1:Theta1_colCount]
Theta2_colCount = Theta2.shape[1]
Theta2_x = Theta2[:,1:Theta2_colCount]
# 正则化向theta^2
term =
np.dot(np.transpose(np.vstack((Theta1_x.reshape(-1,1),Theta2_x.reshape(-1,1)))),np.vstack((Theta1_x.reshape(-1,1),Theta2_x.reshape(-1,1)))))

'''正向传播，每次需要补上一列1的偏置bias'''
a1 = np.hstack((np.ones((m,1)),X))
z2 = np.dot(a1,np.transpose(Theta1))
a2 = sigmoid(z2)
a2 = np.hstack((np.ones((m,1)),a2))
z3 = np.dot(a2,np.transpose(Theta2))
h = sigmoid(z3)
'''代价'''
J = -(np.dot(np.transpose(class_y.reshape(-1,1)),np.log(h.reshape(-1,1)))+np.dot(np.transpose(1-class_y.reshape(-1,1)),np.log(1-h.reshape(-1,1)))-Lambda*term/2)/m

return np.ravel(J)

```

## 4、反向传播BP

上面正向传播可以计算得到 $J(\theta)$ , 使用梯度下降法还需要求它的梯度

BP反向传播的目的就是求代价函数的梯度

假设4层的神经网络,  $\delta_j^{(l)}$  记为-->l层第j个单元的误差

$$\delta_j^{(4)} = a_j^{(4)} - y_i \quad \text{《==》} \quad \delta^{(4)} = a^{(4)} - y \quad (\text{向量化})$$

$$\delta^{(3)} = (\theta^{(3)})^T \delta^{(4)} * g(a^{(3)})$$

$$\delta^{(2)} = (\theta^{(2)})^T \delta^{(3)} * g(a^{(2)})$$

没有  $\delta^{(1)}$ ，因为对于输入没有误差

因为S型函数  $g(z)$  的倒数为：

$$g(z) = g(z)(1 - g(z)),$$

所以上面的  $g(a^{(3)})$  和  $g(a^{(2)})$  可以在前向传播中计算出来

反向传播计算梯度的过程为：

$$\Delta_{ij}^{(l)} = 0 \quad (\Delta \text{是大写的 } \delta)$$

for i=1:m:-

$$a^{(1)} = x^{(i)}$$

-正向传播计算  $a^{(l)}$  ( $l=2, 3, 4, \dots, L$ )

-反向计算  $\delta^{(L)}$ 、 $\delta^{(L-1)}$ ... $\delta^{(2)}$ ；

$$\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta^{(l+1)}$$

$$D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \theta_{ij}^l \quad (j \neq 0)$$

$$D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \theta_{ij}^l \quad j = 0$$

$$\frac{\partial J(\Theta)}{\partial \Theta_{ij}^{(l)}} = D_{ij}^{(l)}$$

最后  $\frac{\partial J(\Theta)}{\partial \Theta_{ij}^{(l)}}$ ，即得到代价函数的梯度

实现代码：

# 梯度

```
def nnGradient(nn_params,input_layer_size,hidden_layer_size,num_labels,X,y,Lambda):
    length = nn_params.shape[0]
    Theta1 = nn_params[0:hidden_layer_size*
(input_layer_size+1)].reshape(hidden_layer_size,input_layer_size+1)
    Theta2 = nn_params[hidden_layer_size*
(input_layer_size+1):length].reshape(num_labels,hidden_layer_size+1)
    m = X.shape[0]
    class_y = np.zeros((m,num_labels))    # 数据的y对应0-9，需要映射为0/1的关系
    # 映射y
    for i in range(num_labels):
        class_y[:,i] = np.int32(y==i).reshape(1,-1) # 注意reshape(1,-1)才可以赋值
```

'''去掉theta1和theta2的第一列，因为正则化时从1开始'''

```
Theta1_colCount = Theta1.shape[1]
Theta1_x = Theta1[:,1:Theta1_colCount]
Theta2_colCount = Theta2.shape[1]
Theta2_x = Theta2[:,1:Theta2_colCount]
```

Theta1\_grad = np.zeros((Theta1.shape)) #第一层到第二层的权重

Theta2\_grad = np.zeros((Theta2.shape)) #第二层到第三层的权重

Theta1[:,0] = 0;

Theta2[:,0] = 0;

'''正向传播，每次需要补上一列1的偏置bias'''

```
a1 = np.hstack((np.ones((m,1)),X))
z2 = np.dot(a1,np.transpose(Theta1))
a2 = sigmoid(z2)
a2 = np.hstack((np.ones((m,1)),a2))
z3 = np.dot(a2,np.transpose(Theta2))
h = sigmoid(z3)
```

'''反向传播，delta为误差，'''

```
delta3 = np.zeros((m,num_labels))
delta2 = np.zeros((m,hidden_layer_size))
for i in range(m):
    delta3[i,:] = h[i,:]-class_y[i,:]
    Theta2_grad = Theta2_grad+np.dot(np.transpose(delta3[i,:].reshape(1,-1)),a2[i,:].reshape(1,-1))
    delta2[i,:] = np.dot(delta3[i,:].reshape(1,-1),Theta2_x)*sigmoidGradient(z2[i,:])
    Theta1_grad = Theta1_grad+np.dot(np.transpose(delta2[i,:].reshape(1,-1)),a1[i,:].reshape(1,-1))
```

```
'''梯度'''
grad =
(np.vstack((Theta1_grad.reshape(-1,1),Theta2_grad.reshape(-1,1)))+Lambda*np.vstack((Theta1.reshape(-1,1),Theta2.reshape(-1,1))))/m
return np.ravel(grad)
```

## 5、BP可以求梯度的原因

实际是利用了链式求导法则

因为下一层的单元利用上一层的单元作为输入进行计算

大体的推导过程如下，最终我们是想预测函数与已知的y非常接近，求均方差的梯度沿着此梯度方向可使代价函数最小化。可对照上面求梯度的过程。

令均方误差  $E_k = \frac{1}{2} \sum_{j=1}^{S_L} (h_\theta(x) - y_j)^2$

令  $\bar{z}^{(l)} = \begin{pmatrix} \theta^{(l-1)} \\ \alpha^{(l-1)} \end{pmatrix}$

$\text{grad} = \frac{\partial E_k}{\partial \theta^{(l-1)}} = \frac{\partial E_k}{\partial z^{(l-1)}} \cdot \frac{\partial z^{(l-1)}}{\partial \theta^{(l-1)}}$

记  $\delta^{(l)} = \frac{\partial E_k}{\partial z^{(l)}} = \frac{\partial E_k}{\partial y} \cdot \frac{\partial y}{\partial z^{(l)}} = (h_\theta(x) - y) \cdot f'(z^{(l)})$

$\therefore \delta^{(l-1)} = \frac{\partial E_k}{\partial z^{(l-1)}} = \frac{\partial \frac{1}{2} \sum_{j=1}^{S_L} (h_\theta(x) - y_j)^2}{\partial z^{(l-1)}} = \frac{1}{2} \sum_{j=1}^{S_L} \frac{\partial (h_\theta(x) - y_j)^2}{\partial z^{(l-1)}}$

$= \frac{1}{2} \sum_{j=1}^{S_L} \frac{\partial (y_j - f(z_j^{(l)}))^2}{\partial z^{(l-1)}} = \sum_{j=1}^{S_L} (f(z_j^{(l)}) - y_j) \cdot \frac{\partial f(z_j^{(l)})}{\partial z^{(l-1)}}$

$= \underbrace{\sum_{j=1}^{S_L} (f(z_j^{(l)}) - y_j) \cdot \frac{\partial f(z_j^{(l)})}{\partial z^{(l-1)}}}_{\delta^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial z^{(l-1)}}$

$= \delta^{(l)} \cdot \frac{\partial z^{(l)}}{\partial z^{(l-1)}}$

$= \delta^{(l)} \cdot \frac{\partial f(z^{(l-1)}) \cdot \theta^{(l-1)}}{\partial z^{(l-1)}}$

$= \underbrace{\delta^{(l)} \cdot \theta^{(l-1)} \cdot f'(z^{(l-1)})}_{\alpha^{(l)}}$

$\Rightarrow \text{grad} = \delta^{(l-1)} \cdot \alpha^{(l-1)} = \alpha^{(l)} \cdot (\delta^{(l)} \cdot \theta^{(l-1)} \cdot f'(z^{(l-1)}))$

求误差更详细的推导过程：

$$\begin{aligned}
\delta_i^{(n_l-1)} &= \frac{\partial}{\partial z_i^{n_l-1}} J(W, b; x, y) = \frac{\partial}{\partial z_i^{n_l-1}} \frac{1}{2} \|y - h_{W,b}(x)\|^2 = \frac{\partial}{\partial z_i^{n_l-1}} \frac{1}{2} \sum_{j=1}^{S_{n_l}} (y_j - a_j^{(n_l)})^2 \\
&= \frac{1}{2} \sum_{j=1}^{S_{n_l}} \frac{\partial}{\partial z_i^{n_l-1}} (y_j - a_j^{(n_l)})^2 = \frac{1}{2} \sum_{j=1}^{S_{n_l}} \frac{\partial}{\partial z_i^{n_l-1}} (y_j - f(z_j^{(n_l)}))^2 \\
&= \sum_{j=1}^{S_{n_l}} -(y_j - f(z_j^{(n_l)})) \cdot \frac{\partial}{\partial z_i^{(n_l-1)}} f(z_j^{(n_l)}) = \sum_{j=1}^{S_{n_l}} -(y_j - f(z_j^{(n_l)})) \cdot f'(z_j^{(n_l)}) \cdot \frac{\partial z_j^{(n_l)}}{\partial z_i^{(n_l-1)}} \\
&= \sum_{j=1}^{S_{n_l}} \delta_j^{(n_l)} \cdot \frac{\partial z_j^{(n_l)}}{\partial z_i^{(n_l-1)}} = \sum_{j=1}^{S_{n_l}} \left( \delta_j^{(n_l)} \cdot \frac{\partial}{\partial z_i^{(n_l-1)}} \sum_{k=1}^{S_{n_l-1}} f(z_k^{(n_l-1)}) \cdot W_{jk}^{n_l-1} \right) \\
&= \sum_{j=1}^{S_{n_l}} \delta_j^{(n_l)} \cdot W_{ji}^{n_l-1} \cdot f'(z_i^{n_l-1}) = \left( \sum_{j=1}^{S_{n_l}} W_{ji}^{n_l-1} \delta_j^{(n_l)} \right) f'(z_i^{n_l-1})
\end{aligned}$$

## 6、梯度检查

检查利用BP求的梯度是否正确

利用导数的定义验证：

求出来的数值梯度应该与BP求出的梯度非常接近

验证BP正确后就不再需要再执行验证梯度的算法了

实现代码：

```

# 检验梯度是否计算正确
# 检验梯度是否计算正确
def checkGradient(Lambda = 0):
    '''构造一个小型的神经网络验证，因为数值法计算梯度很浪费时间，而且验证正确后之后就不再需要验证了'''
    input_layer_size = 3
    hidden_layer_size = 5
    num_labels = 3
    m = 5
    initial_Theta1 = debugInitializeWeights(input_layer_size,hidden_layer_size);
    initial_Theta2 = debugInitializeWeights(hidden_layer_size,num_labels)
    X = debugInitializeWeights(input_layer_size-1,m)

```

```

y = 1+np.transpose(np.mod(np.arange(1,m+1), num_labels))# 初始化y

y = y.reshape(-1,1)
nn_params = np.vstack((initial_Theta1.reshape(-1,1),initial_Theta2.reshape(-1,1))) #展开theta
'''BP求出梯度'''
grad = nnGradient(nn_params, input_layer_size, hidden_layer_size,
                   num_labels, X, y, Lambda)
'''使用数值法计算梯度'''
num_grad = np.zeros((nn_params.shape[0]))
step = np.zeros((nn_params.shape[0]))
e = 1e-4
for i in range(nn_params.shape[0]):
    step[i] = e
    loss1 = nnCostFunction(nn_params-step.reshape(-1,1), input_layer_size, hidden_layer_size,
                           num_labels, X, y,
                           Lambda)
    loss2 = nnCostFunction(nn_params+step.reshape(-1,1), input_layer_size, hidden_layer_size,
                           num_labels, X, y,
                           Lambda)
    num_grad[i] = (loss2-loss1)/(2*e)
    step[i]=0
# 显示两列比较
res = np.hstack((num_grad.reshape(-1,1),grad.reshape(-1,1)))
print res

```

## 7、权重的随机初始化

神经网络不能像逻辑回归那样初始化theta为0,因为若是每条边的权重都为0，每个神经元都是相同的输出，在反向传播中也会得到同样的梯度，最终只会预测一种结果。

所以应该初始化为接近0的数

实现代码

```

# 随机初始化权重theta
def randInitializeWeights(L_in,L_out):
    W = np.zeros((L_out,1+L_in)) # 对应theta的权重
    epsilon_init = (6.0/(L_out+L_in))**0.5

```

```

W = np.random.rand(L_out,1+L_in)*2*epsilon_init-epsilon_init # np.random.rand(L_out,1+L_in)产生L_out*(1+L_in)大小的随机矩阵
return W

```

## 8、预测

正向传播预测结果

实现代码

```

# 预测
def predict(Theta1,Theta2,X):
    m = X.shape[0]
    num_labels = Theta2.shape[0]
    #p = np.zeros((m,1))
    '''正向传播，预测结果'''
    X = np.hstack((np.ones((m,1)),X))
    h1 = sigmoid(np.dot(X,np.transpose(Theta1)))
    h1 = np.hstack((np.ones((m,1)),h1))
    h2 = sigmoid(np.dot(h1,np.transpose(Theta2)))

    ...
    返回h中每一行最大值所在的列号
    - np.max(h, axis=1)返回h中每一行的最大值（是某个数字的最大概率）
    - 最后where找到的最大概率所在的列号（列号即是对应的数字）
    ...
    #np.savetxt("h2.csv",h2,delimiter=',')
    p = np.array(np.where(h2[0,:] == np.max(h2, axis=1)[0]))
    for i in np.arange(1, m):
        t = np.array(np.where(h2[i,:] == np.max(h2, axis=1)[i]))
        p = np.vstack((p,t))
    return p

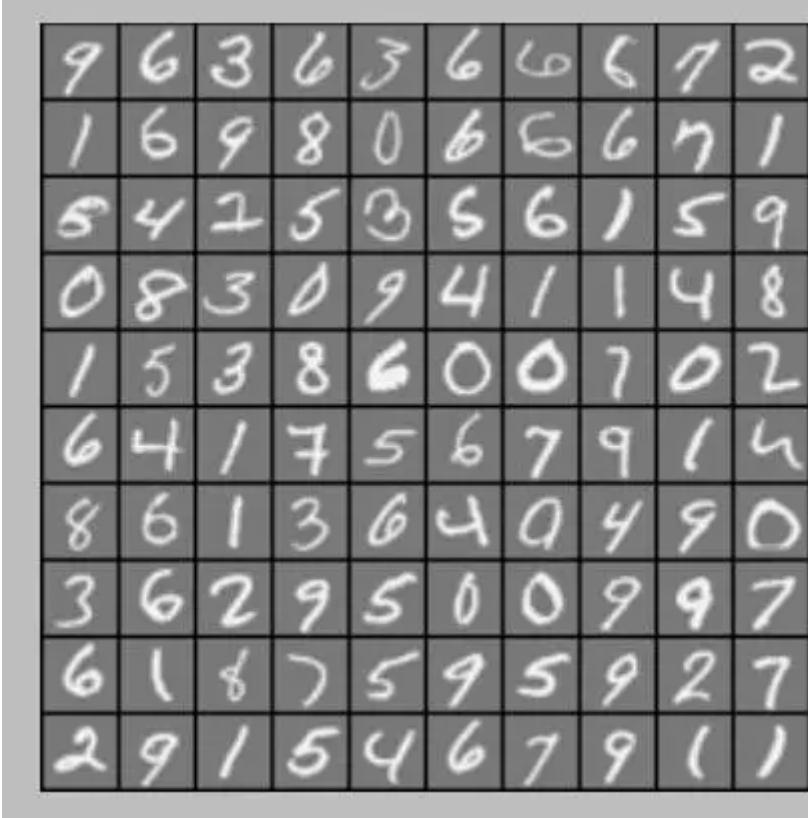
```

## 9、输出结果

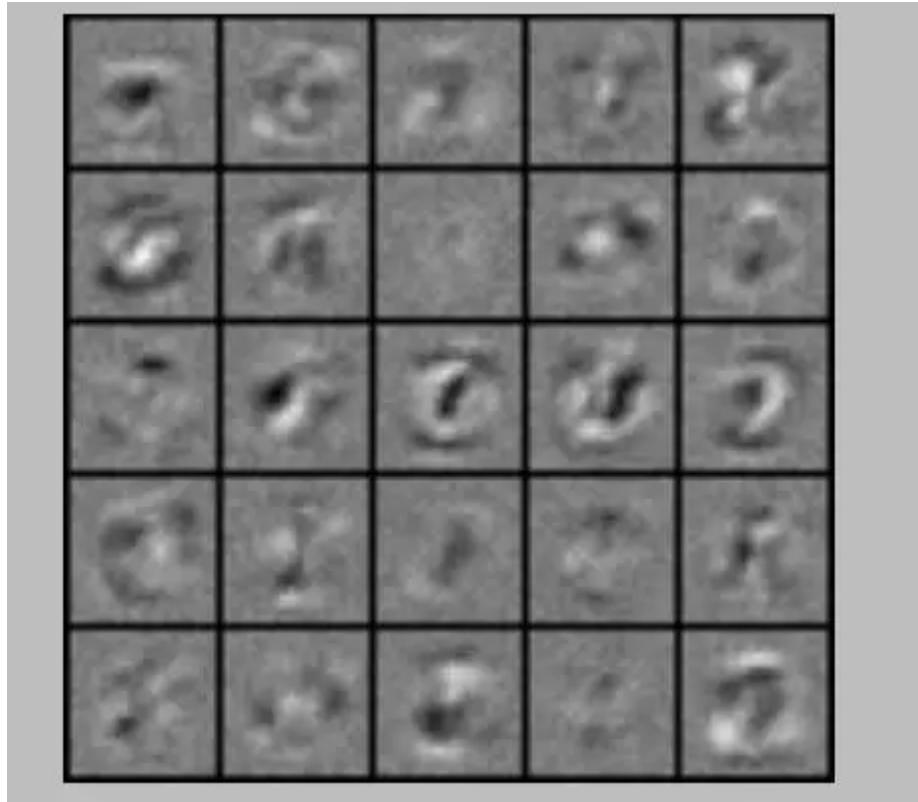
梯度检查：

```
NeuralNetwork.py (j ▾) Debug I/O (stdin, stdout, stderr) appears below
[[ 2.06735785e-02  2.06735785e-02]
 [ -3.32326278e-05 -3.32326269e-05]
 [ 2.16724509e-04  2.16724507e-04]
 [ 2.67426130e-04  2.67426128e-04]
 [ 6.27845489e-03  6.27845489e-03]
 [ -4.86603025e-05 -4.86603041e-05]
 [ 1.01842170e-04  1.01842170e-04]
 [ 1.58711422e-04  1.58711422e-04]
 [ -1.38881861e-02 -1.38881861e-02]
 [ -1.93447791e-05 -1.93447781e-05]
 [ -1.06655433e-04 -1.06655434e-04]
 [ -9.59075774e-05 -9.59075753e-05]
 [ -2.12864353e-02 -2.12864353e-02]
 [ 2.77471823e-05  2.77471821e-05]
 [ -2.17113305e-04 -2.17113306e-04]
 [ -2.62360820e-04 -2.62360822e-04]
 [ -9.11393890e-03 -9.11393891e-03]
 [ 4.93355845e-05  4.93355833e-05]
 [ -1.27947126e-04 -1.27947125e-04]
 [ -1.87595837e-04 -1.87595837e-04]
 [ 4.88199720e-01  4.88199720e-01]
 [ 2.44197480e-01  2.44197480e-01]
 [ 2.44300580e-01  2.44300580e-01]
 [ 2.44300580e-01  2.44300580e-01]]
```

随机显示100个手写数字



显示theta1权重



训练集预测准确度

```

Current function value: 0.829167
Iterations: 21
Function evaluations: 75
Gradient evaluations: 63
39.8369998932
[-0.47325389 -0.07951369  0.0572995  ..., -0.20159648  0.9141755
 -2.02774864]
预测准确度为: 91.120000%

```

归一化后训练集预测准确度

```

[-0.30453506  0.10040848 -0.04501221  ..., -1.45447913 -0.12299637
 1.19577227]
预测准确度为: 98.020000%

```

## 四、SVM支持向量机

### 1、代价函数

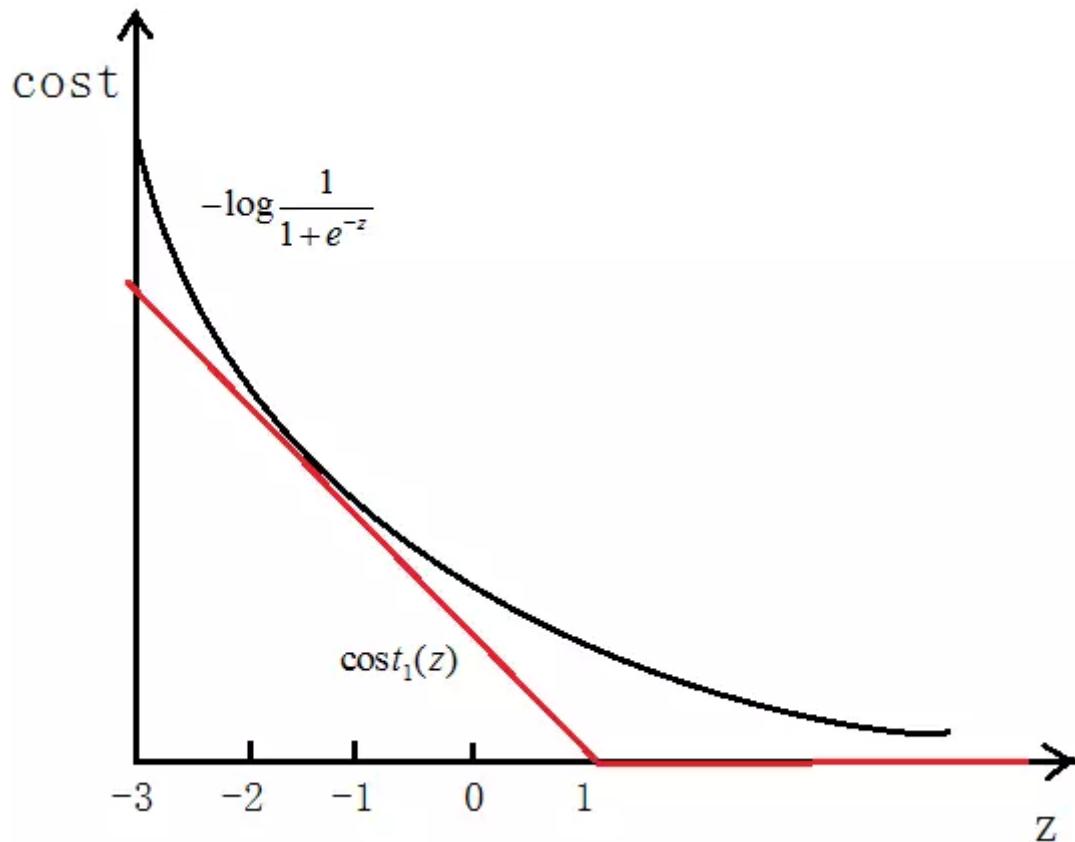
在逻辑回归中，我们的代价为：

$$\text{cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & y=1 \\ -\log(1-h_\theta(x)) & y=0 \end{cases}$$

其中：

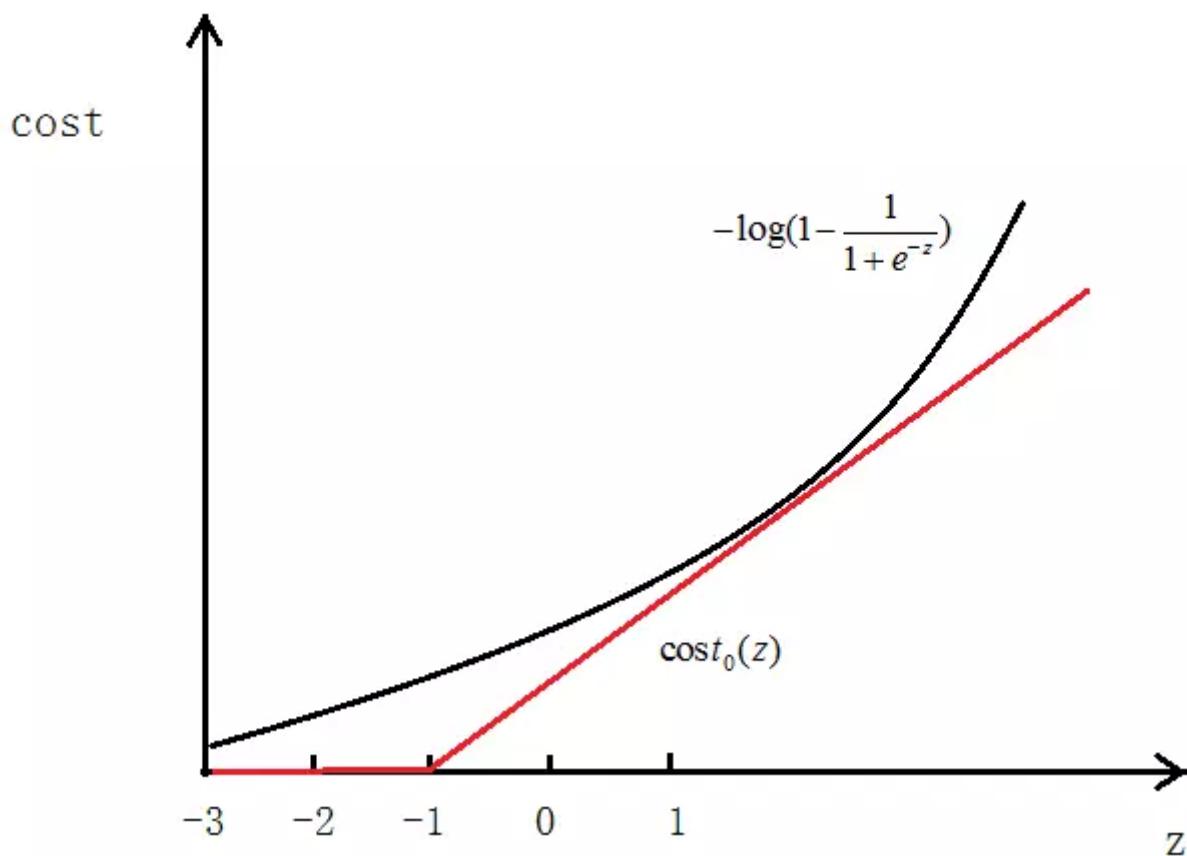
$$h_\theta(z) = \frac{1}{1+e^{-z}}, z = \theta^T x$$

如图所示，如果y=1，cost代价函数如图所示



我们想让  $\theta^T x >> 0$ ，即  $z >> 0$ ，这样的话 cost 代价函数才会趋于最小（这是我们想要的），所以用途中红色的函数  $\text{cost}_1(z)$  代替逻辑回归中的 cost

当  $y=0$  时同样，用  $\text{cost}_0(z)$  代替



最终得到的代价函数为：

$$J(\theta) = C \sum_{i=1}^m [y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)}) \text{cost}_0(\theta^T x^{(i)})] + \frac{1}{2} \sum_{j=1}^n \theta_j^2$$

最后我们想要  $\min_{\theta} J(\theta)$

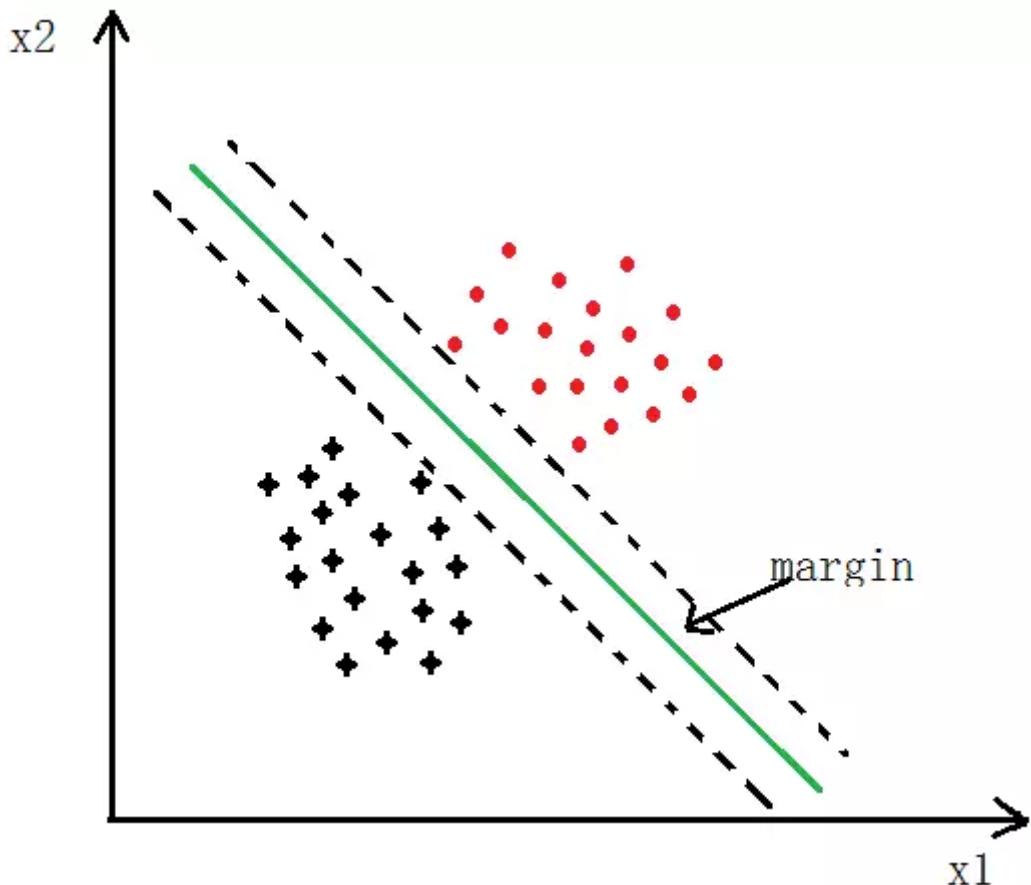
之前我们逻辑回归中的代价函数为：

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_{\theta}(x^{(i)}) + (1-y^{(i)}) \log(1-h_{\theta}(x^{(i)}))] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

可以认为这里的  $C = \frac{m}{\lambda}$ ，只是表达形式问题，这里C的值越大，SVM的决策边界的margin也越大，下面会说明

## 2、Large Margin

如下图所示，SVM分类会使用最大的margin将其分开



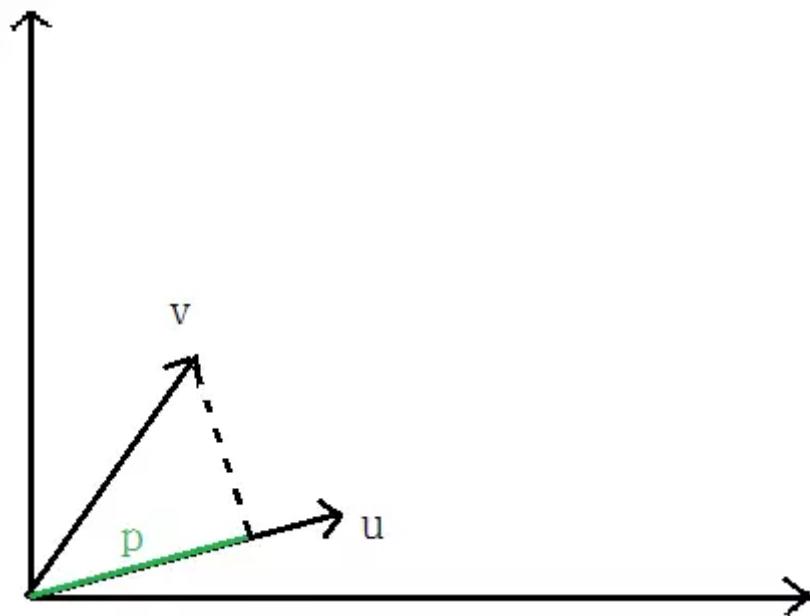
先说一下向量内积

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

$\|u\|$  表示  $u$  的欧几里得范数 ( 欧式范数 ) ,  $\|u\| = \sqrt{u_1^2 + u_2^2}$

向量  $v$  在向量  $u$  上的投影的长度记为  $p$  , 则 : 向量内积 :

$$u^T v = p \|u\| = u_1 v_1 + u_2 v_2$$



$$\cos \theta = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}| |\vec{v}|}$$

根据向量夹角公式推导一下即可 ,

前面说过 , 当  $C$  越大时 , margin 也就越大 , 我们的目的是最小化代价函数  $J(\theta)$  , 当 margin 最大时 ,  $C$  的乘积项

$$\sum_{i=1}^m [y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T x^{(i)})]$$

要很小 , 所以近似为 :

$$J(\theta) = C0 + \frac{1}{2} \sum_{j=1}^n \theta_j^2 = \frac{1}{2} \sum_{j=1}^n \theta_j^2 = \frac{1}{2} (\theta_1^2 + \theta_2^2) = \frac{1}{2} \sqrt{\theta_1^2 + \theta_2^2}$$

我们最后的目的就是求使代价最小的 $\theta$

由

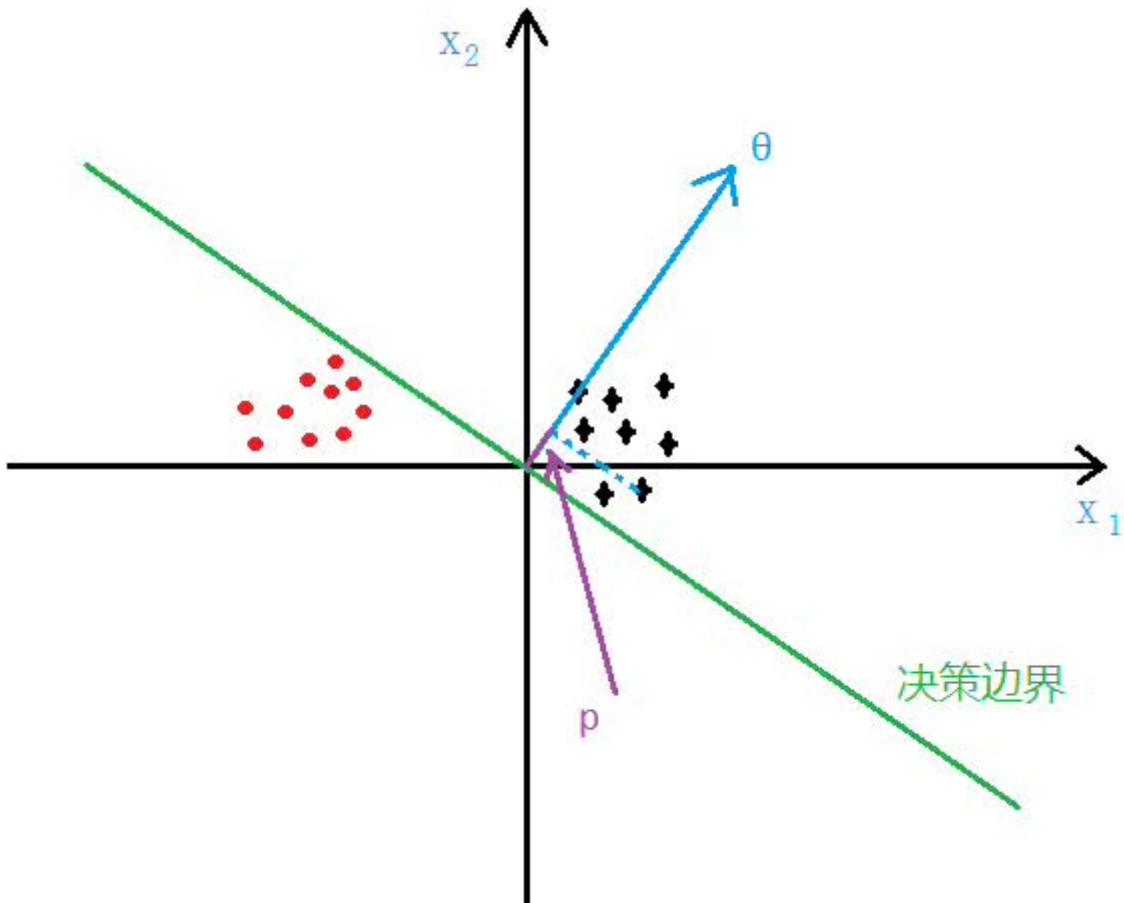
$$\begin{cases} \theta^T x^{(i)} \geq 1 & (y^{(i)} = 1) \\ \theta^T x^{(i)} \leq -1 & (y^{(i)} = 0) \end{cases}$$

可以得到：

$$\begin{cases} p^{(i)} \|\theta\| \geq 1 & (y^{(i)} = 1) \\ p^{(i)} \|\theta\| \leq -1 & (y^{(i)} = 0) \end{cases}$$

$p$ 即为 $x$ 在 $\theta$ 上的投影

如下图所示，假设决策边界如图，找其中的一个点，到 $\theta$ 上的投影为 $p$ ，则 $p \|\theta\| \geq 1$ 或者 $p \|\theta\| \leq -1$ ，若是 $p$ 很小，则需要 $\|\theta\|$ 很大，这与我们要求的 $\theta$ 使 $\|\theta\| = \frac{1}{2} \sqrt{\theta_1^2 + \theta_2^2}$ 最小相违背，所以最后求的是large margin



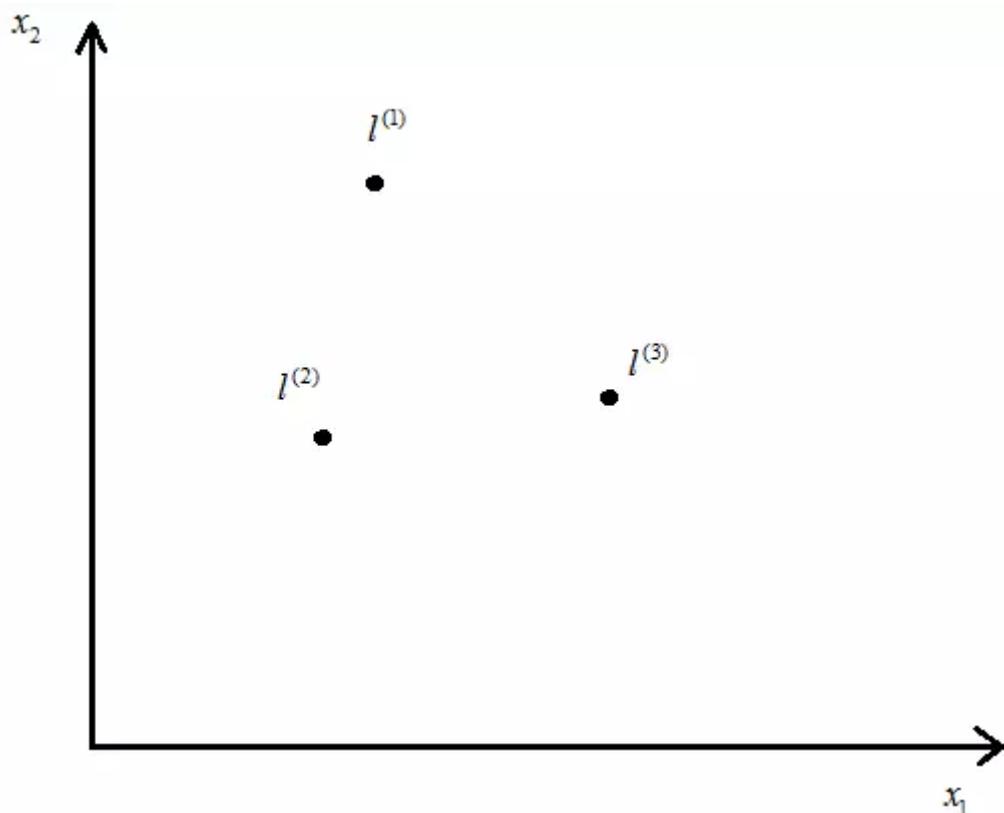
### 3、SVM Kernel (核函数)

对于线性可分的问题，使用线性核函数即可

对于线性不可分的问题，在逻辑回归中，我们是将 feature 映射为使用多项式的形式  
，SVM中也有多项式核函数，但是更常用的是高斯核函数，也称为 RBF核

$$\text{高斯核函数为 : } f(x) = e^{-\frac{\|x-u\|^2}{2\sigma^2}}$$

假设如图几个点，

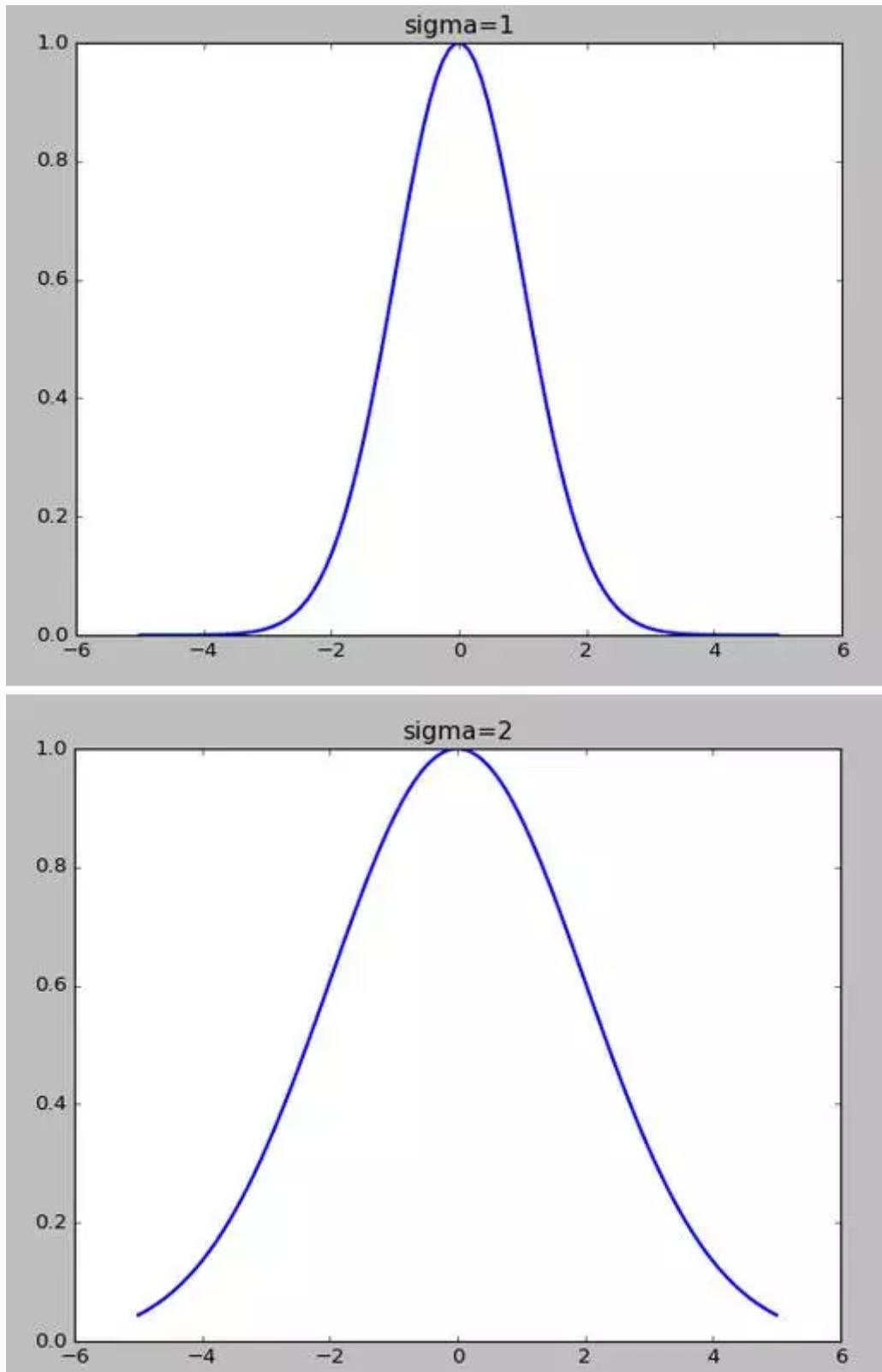


令：

...

可以看出，若是 $x$ 与  $l^{(1)}$  距离较近， $\Rightarrow$   $f(x)$  值较大，（即相似度较大），若是 $x$ 与  $l^{(3)}$  距离较远， $\Rightarrow$   $f(x)$  值较小，（即相似度较低）

高斯核函数的 $\sigma$ 越小， $f$ 下降的越快



如何选择初始的  $l^{(1)} l^{(2)} l^{(3)} \dots$

训练集 :  $((x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)}))$

选择 :  $l^{(1)} = x^{(1)}, l^{(2)} = x^{(2)}, \dots, l^{(m)} = x^{(m)}$

对于给出的  $x$ , 计算  $f$ , 令 :  $f_0^{(i)} = 1$ ,

所以： $f^{(i)} \in R^{m+1}$

最小化J求出 $\theta$ ，

$$J(\theta) = C \sum_{i=1}^m [y^{(i)} \text{cost}_1(\theta^T f^{(i)}) + (1-y^{(i)}) \text{cost}_0(\theta^T f^{(i)})] + \frac{1}{2} \sum_{j=1}^n \theta_j^2$$

如果  $\theta^T f \geq 0$ , ==> 预测  $y=1$

#### 4. 使用scikit-learn中的SVM模型代码

全部代码

[https://github.com/lawlite19/MachineLearning\\_Python/blob/master/SVM/SVM\\_scikit-learn.py](https://github.com/lawlite19/MachineLearning_Python/blob/master/SVM/SVM_scikit-learn.py)

线性可分的,指定核函数为linear：

```
'''data1——线性分类'''
data1 = spio.loadmat('data1.mat')
X = data1['X']
y = data1['y']
y = np.ravel(y)
plot_data(X,y)

model = svm.SVC(C=1.0,kernel='linear').fit(X,y) # 指定核函数为线性核函数
```

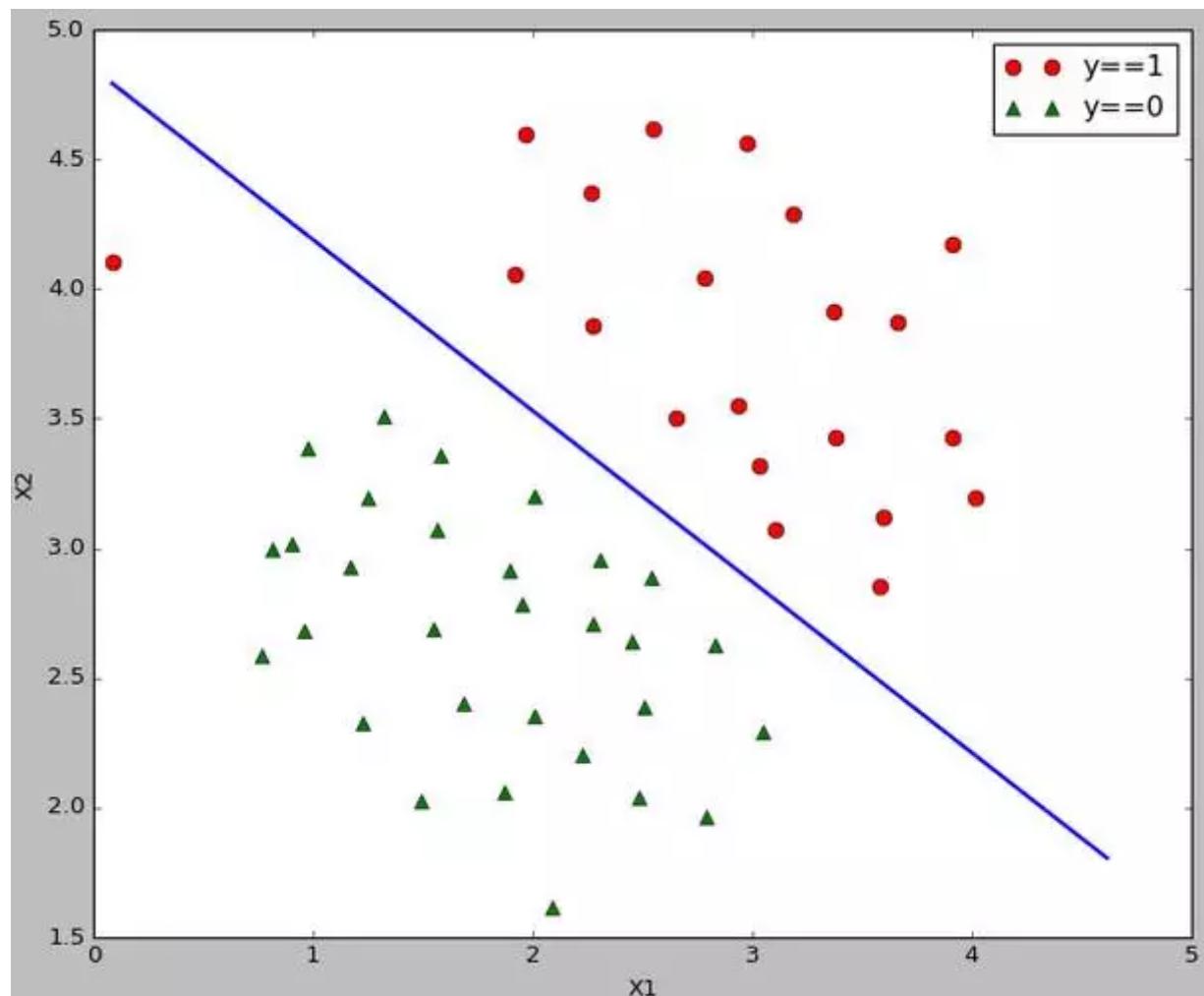
非线性可分的，默认核函数为rbf

```
'''data2——非线性分类'''
data2 = spio.loadmat('data2.mat')
X = data2['X']
y = data2['y']
y = np.ravel(y)
plt = plot_data(X,y)
plt.show()

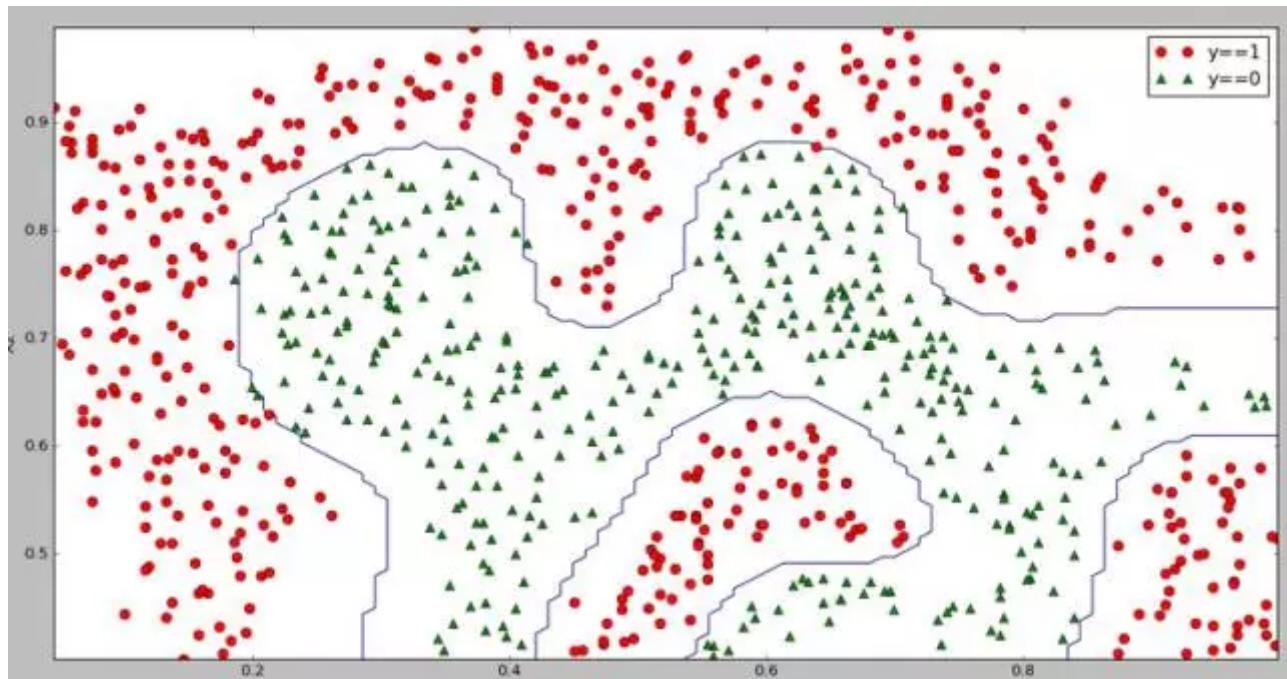
model = svm.SVC(gamma=100).fit(X,y) # gamma为核函数的系数，值越大拟合的越好
```

## 5、运行结果

线性可分的决策边界：



线性不可分的决策边界：



## 五、K-Means聚类算法

全部代码

[https://github.com/lawlite19/MachineLearning\\_Python/blob/master/K-Means/K-Menas.py](https://github.com/lawlite19/MachineLearning_Python/blob/master/K-Means/K-Menas.py)

### 1、聚类过程

聚类属于无监督学习，不知道y的标记分为K类

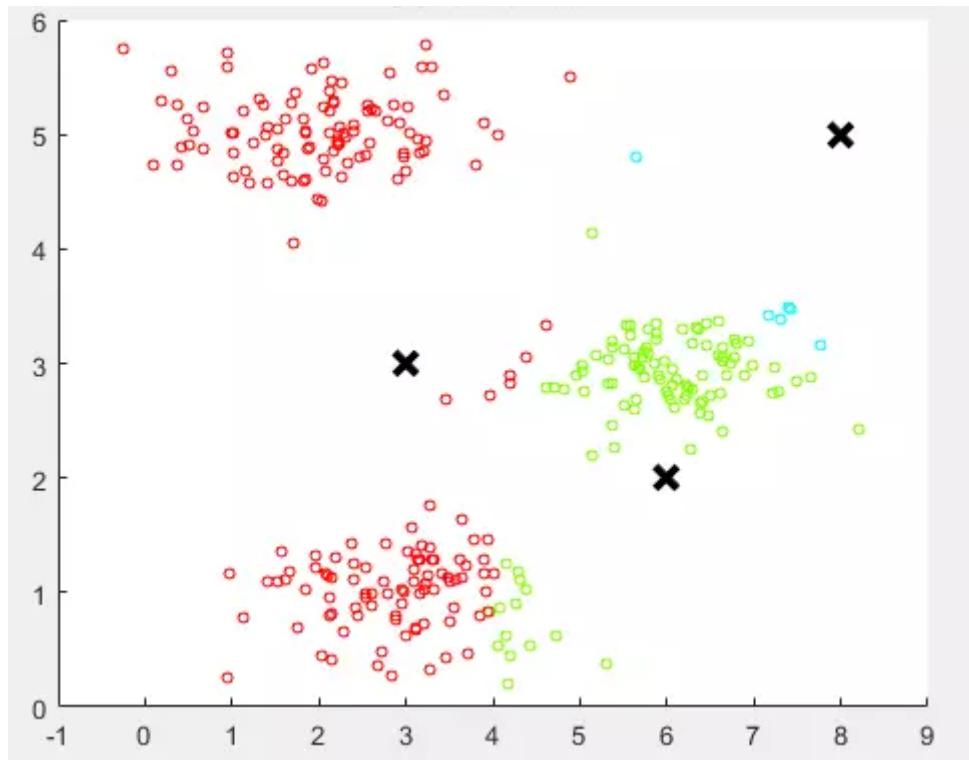
K-Means算法分为两个步骤

第一步：簇分配，随机选K个点作为中心，计算到这K个点的距离，分为K个簇

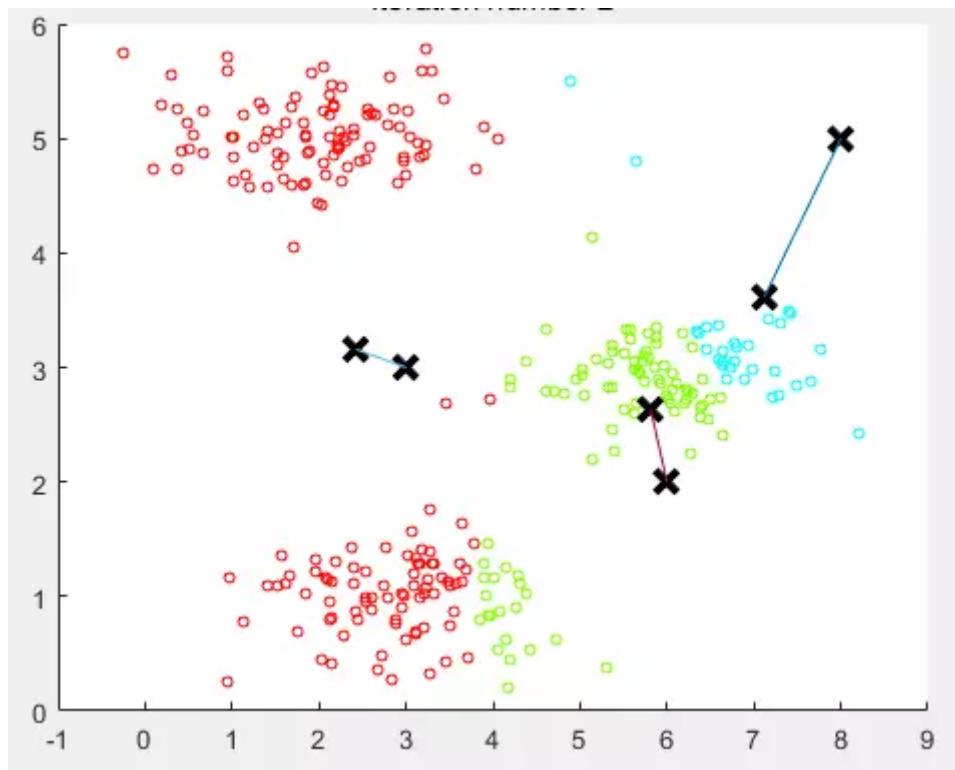
第二步：移动聚类中心：重新计算每个簇的中心，移动中心，重复以上步骤。

如下图所示：

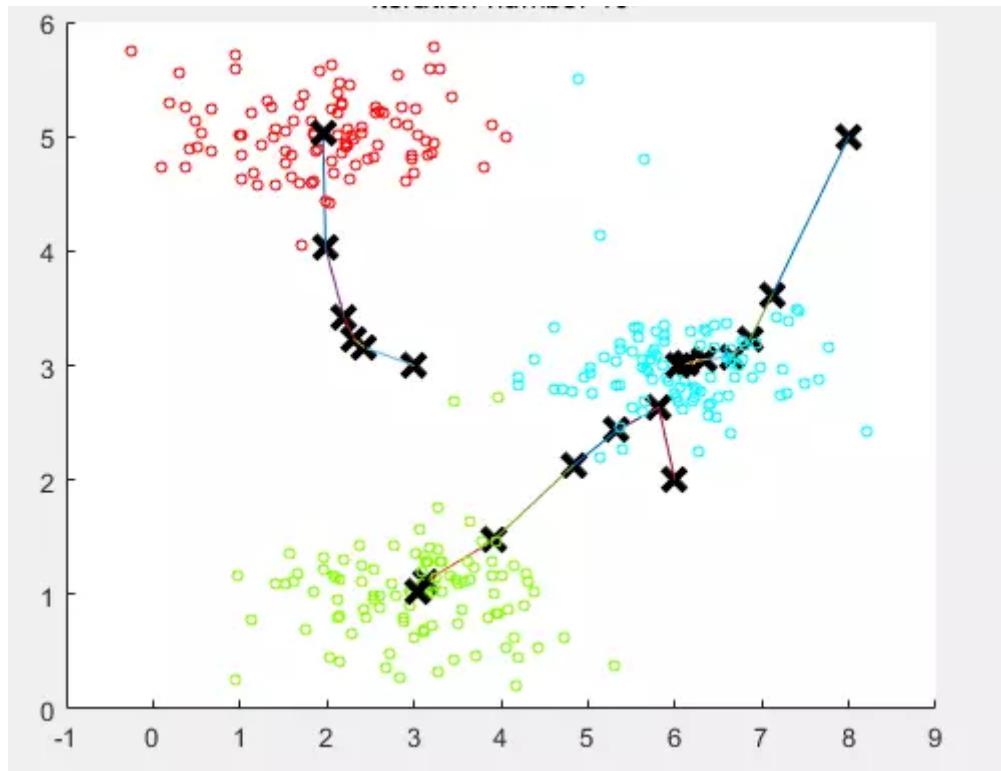
随机分配的聚类中心



重新计算聚类中心，移动一次



最后10步之后的聚类中心



计算每条数据到哪个中心最近实现代码：

```
# 找到每条数据距离哪个类中心最近
def findClosestCentroids(X,initial_centroids):
    m = X.shape[0]          # 数据条数
    K = initial_centroids.shape[0] # 类的总数
    dis = np.zeros((m,K))      # 存储计算每个点分别到K个类的距离
    idx = np.zeros((m,1))       # 要返回的每条数据属于哪个类

    '''计算每个点到每个类中心的距离'''
    for i in range(m):
        for j in range(K):
            dis[i,j] = np.dot((X[i,:]-initial_centroids[j,:]).reshape(1,-1),(X[i,:]-initial_centroids[j,:]).reshape(-1,1))

    '''返回dis每一行的最小值对应的列号，即为对应的类别
    - np.min(dis, axis=1)返回每一行的最小值
    - np.where(dis == np.min(dis, axis=1).reshape(-1,1)) 返回对应最小值的坐标
    - 注意：可能最小值对应的坐标有多个，where都会找出来，所以返回时返回前m个需要的即可（因为对于
    多个最小值，属于哪个类别都可以）'''
    ...
    dummy,idx = np.where(dis == np.min(dis, axis=1).reshape(-1,1))
    return idx[0:dis.shape[0]] # 注意截取一下
```

计算类中心实现代码：

```
# 计算类中心
def computerCentroids(X, idx, K):
    n = X.shape[1]
    centroids = np.zeros((K, n))
    for i in range(K):
        centroids[i, :] = np.mean(X[np.ravel(idx == i), :], axis=0).reshape(1, -1) # 索引要是一维的, axis=0为每一列, idx==i一次找出属于哪一类的, 然后计算均值
    return centroids
```

## 2、目标函数

也叫做失真代价函数

$$J(c^{(1)}, \dots, c^{(m)}, u_1, \dots, u_k) = \frac{1}{m} \sum_{i=1}^m \|x^{(i)} - u_{c^{(i)}}\|^2$$

$$\min_{\substack{c^{(1)}, \dots, c^{(m)} \\ u_1, \dots, u_k}} J(c^{(1)}, \dots, c^{(m)}, u_1, \dots, u_k)$$

最后我们想得到：

其中  $c^{(i)}$  表示第  $i$  条数据距离哪个类中心最近，其中  $u_i$  即为聚类的中心

## 3、聚类中心的选择

随机初始化，从给定的数据中随机抽取  $K$  个作为聚类中心

随机一次的结果可能不好，可以随机多次，最后取使代价函数最小的作为中心

实现代码：(这里随机一次)

```
# 初始化类中心--随机取K个点作为聚类中心
def kMeansInitCentroids(X, K):
    m = X.shape[0]
    m_arr = np.arange(0, m) # 生成0-m-1
    centroids = np.zeros((K, X.shape[1]))
    np.random.shuffle(m_arr) # 打乱m_arr顺序
```

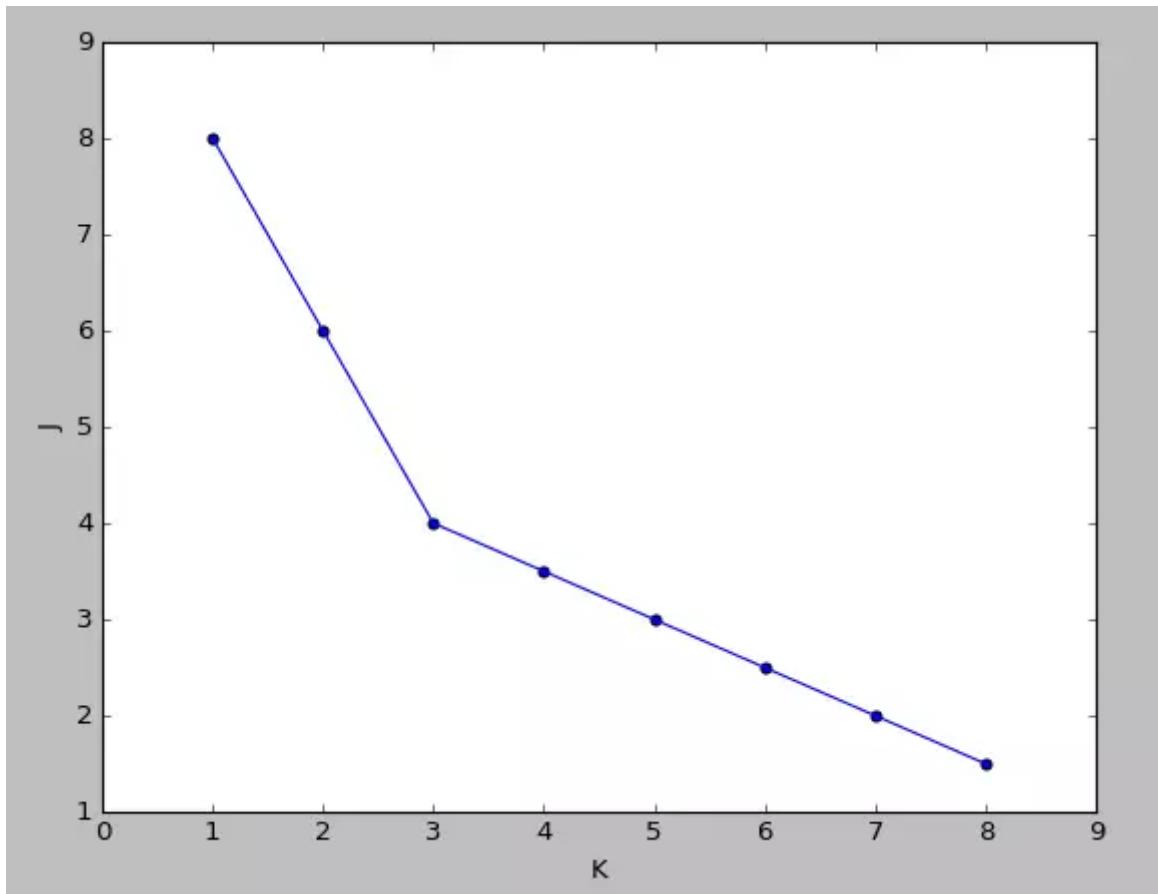
```
rand_indices = m_arr[:K] # 取前K个  
centroids = X[rand_indices,:]  
return centroids
```

## 4、聚类个数K的选择

聚类是不知道y的label的，所以不知道真正的聚类个数

肘部法则 ( Elbow method )

作代价函数J和K的图，若是出现一个拐点，如下图所示，K就取拐点处的值，下图此时K=3



若是很平滑就不明确，人为选择。

第二种就是人为观察选择

## 5、应用——图片压缩

将图片的像素分为若干类，然后用这个类代替原来的像素值

执行聚类的算法代码：

```
# 聚类算法
def runKMeans(X,initial_centroids,max_iters,plot_process):
    m,n = X.shape           # 数据条数和维度
    K = initial_centroids.shape[0] # 类数
    centroids = initial_centroids # 记录当前类中心
    previous_centroids = centroids # 记录上一次类中心
    idx = np.zeros((m,1))       # 每条数据属于哪个类

    for i in range(max_iters):   # 迭代次数
        print u'迭代计算次数 : %d'%(i+1)
        idx = findClosestCentroids(X, centroids)
        if plot_process: # 如果绘制图像
            plt = plotProcessKMeans(X,centroids,previous_centroids) # 画聚类中心的移动过程
            previous_centroids = centroids # 重置
        centroids = computerCentroids(X, idx, K) # 重新计算类中心
    if plot_process: # 显示最终的绘制结果
        plt.show()
    return centroids,idx # 返回聚类中心和数据属于哪个类
```

## 6、使用scikit-learn库中的线性模型实现聚类

[https://github.com/lawlite19/MachineLearning\\_Python/blob/master/K-Means/K-Means\\_scikit-learn.py](https://github.com/lawlite19/MachineLearning_Python/blob/master/K-Means/K-Means_scikit-learn.py)

导入包

```
from sklearn.cluster import KMeans
```

使用模型拟合数据

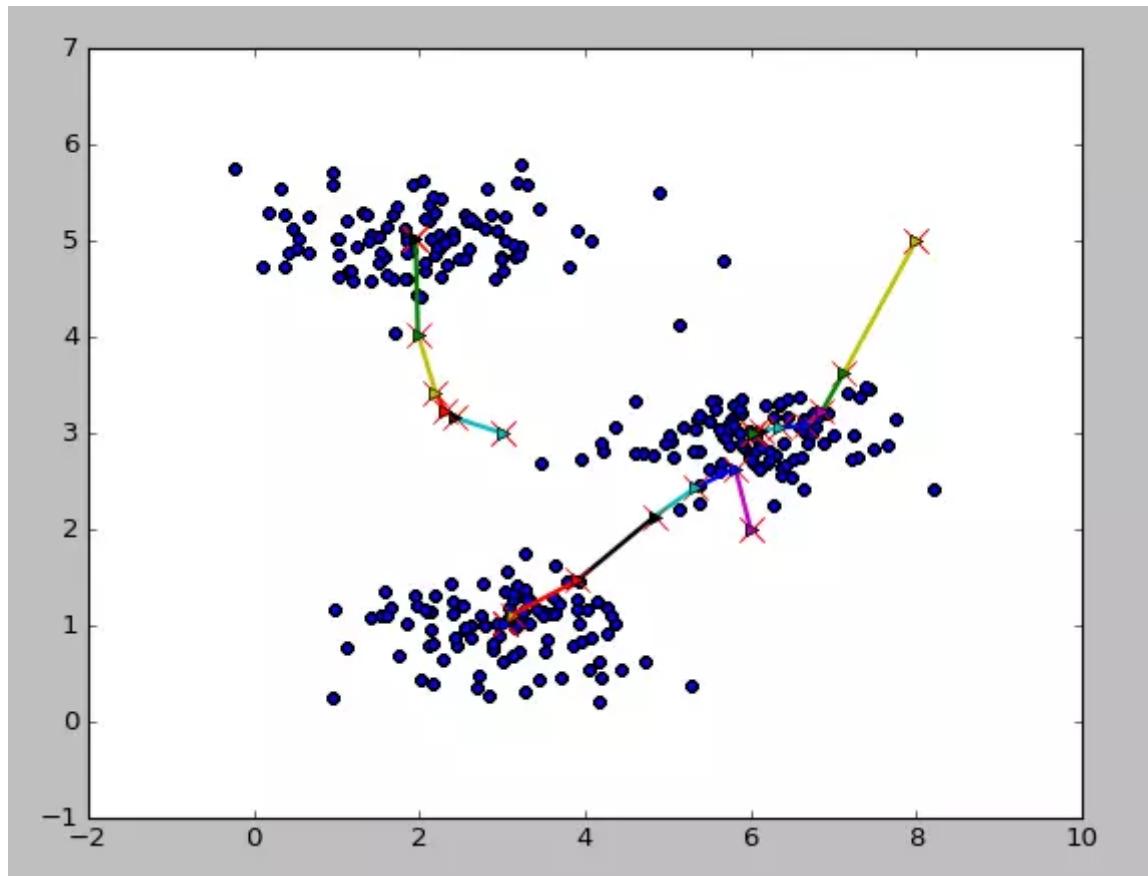
```
model = KMeans(n_clusters=3).fit(X) # n_clusters指定3类，拟合数据
```

聚类中心

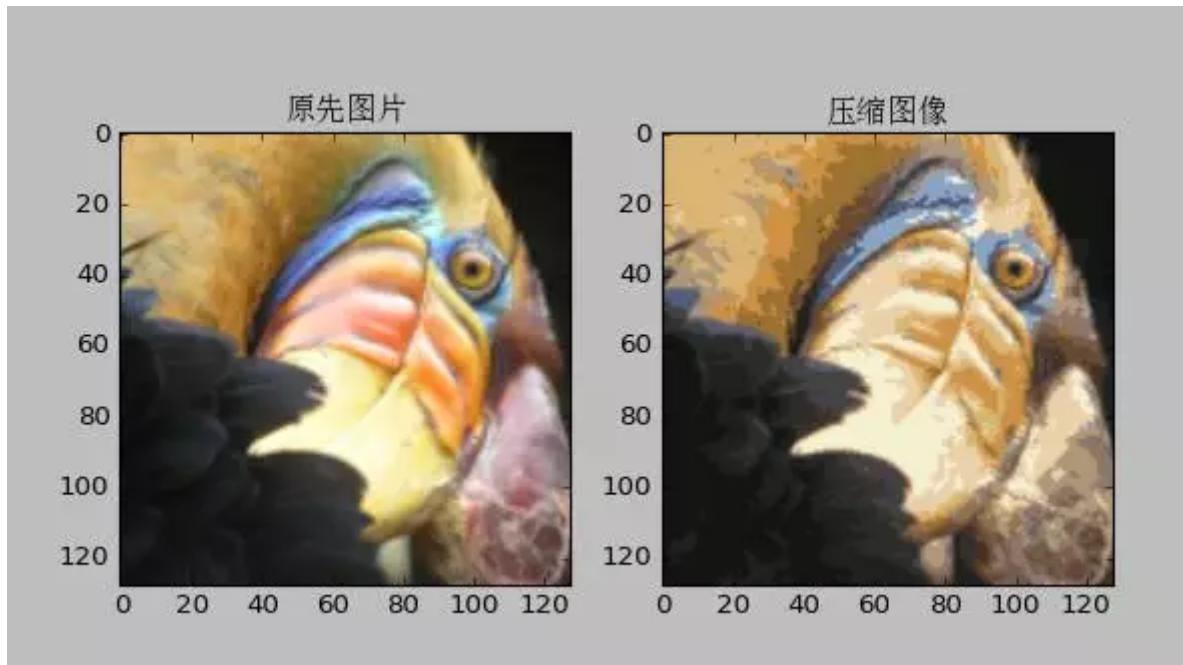
```
centroids = model.cluster_centers_ # 聚类中心
```

## 7、运行结果

二维数据类中心的移动



图片压缩



## 六、PCA主成分分析（降维）

全部代码

[https://github.com/lawlite19/MachineLearning\\_Python/blob/master/PCA/PCA.py](https://github.com/lawlite19/MachineLearning_Python/blob/master/PCA/PCA.py)

## 1、用处

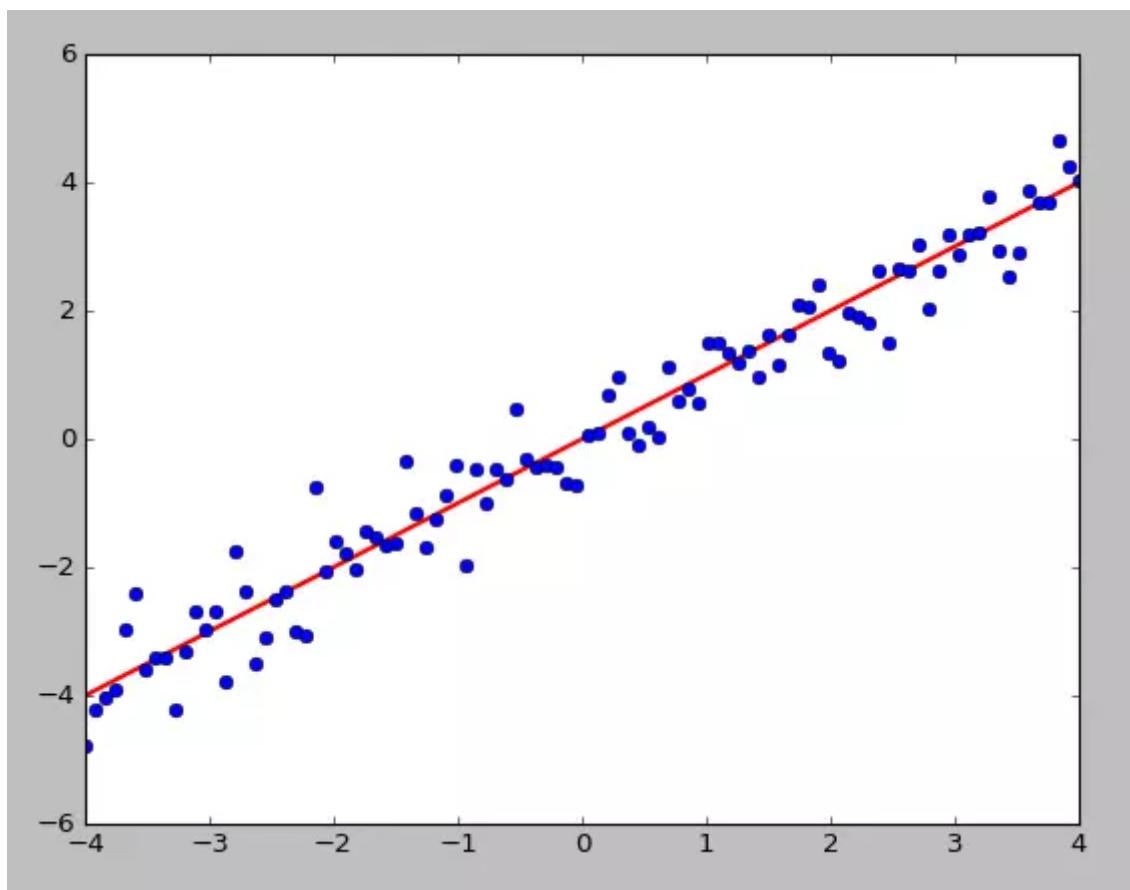
数据压缩 ( Data Compression ), 使程序运行更快

可视化数据 , 例如 3D-->2D 等

.....

## 2、2D-->1D , nD-->kD

如下图所示 , 所有数据点可以投影到一条直线 , 是投影距离的平方和 ( 投影误差 ) 最小



注意数据需要归一化处理

思路是找 1 个向量  $u$ , 所有数据投影到上面使投影距离最小

那么  $nD \rightarrow kD$  就是找  $k$  个向量 ,

所有数据投影到上面使投影误差最小

eg: 3D-->2D, 2个向量 

就代表一个平面了，所有点投影到这个平面的投影误差最小即可

### 3、主成分分析PCA与线性回归的区别

线性回归是找x与y的关系，然后用于预测y

PCA是找一个投影面，最小化data到这个投影面的投影误差

### 4、PCA降维过程

数据预处理（均值归一化）

$$\text{公式 : } \mathbf{x}_j^{(i)} = \frac{\mathbf{x}_j^{(i)} - u_j}{s_j}$$

就是减去对应feature的均值，然后除以对应特征的标准差（也可以是最大值-最小值）

实现代码：

```
# 归一化数据
def featureNormalize(X):
    ''' (每一个数据-当前列的均值) /当前列的标准差'''
    n = X.shape[1]
    mu = np.zeros((1,n));
    sigma = np.zeros((1,n))

    mu = np.mean(X, axis=0)
    sigma = np.std(X, axis=0)
    for i in range(n):
        X[:,i] = (X[:,i]-mu[i])/sigma[i]
    return X, mu, sigma
```

计算协方差矩阵 $\Sigma$  ( Covariance Matrix ) :

$$\Sigma = \frac{1}{m} \sum_{i=1}^n x^{(i)} (x^{(i)})^T$$

注意这里的 $\Sigma$ 和求和符号不同

协方差矩阵对称正定 ( 不理解正定的看看线代 )

大小为 $n \times n$ ,  $n$ 为feature的维度

实现代码 :

```
Sigma = np.dot(np.transpose(X_norm), X_norm)/m # 求Sigma
```

计算 $\Sigma$ 的特征值和特征向量

可以是用svd奇异值分解函数 :  $U, S, V = \text{svd}(\Sigma)$

返回的是与 $\Sigma$ 同样大小的对角阵 $S$  ( 由 $\Sigma$ 的特征值组成 ) [注意 : matlab中函数返回的是对角阵 , 在 python中返回的是一个向量 , 节省空间 ]

还有两个酉矩阵 $U$ 和 $V$  , 且 $\Sigma = U S V^T$

$$U = \begin{bmatrix} & & & \\ u^{(1)} & u^{(2)} & \cdots & u^{(n)} \\ & & & \end{bmatrix} \in R^{n \times n}$$

注意 : svd函数求出的 $S$ 是按特征值降序排列的 , 若不是使用svd, 需要按特征值大小重新排列 $U$

降维

选取 $U$ 中的前 $K$ 列 ( 假设要降为 $K$ 维 )

$$Z^{(i)} = \begin{bmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \cdots & u^{(k)} \\ | & | & & | \end{bmatrix}^T \quad \text{X}^{(i)} = \begin{bmatrix} - & (u^{(1)})^T & - \\ - & (u^{(1)})^T & - \\ \vdots & & \\ - & (u_{\text{kxn}}^{(1)})^T & - \end{bmatrix}_{\text{nx1}} \quad \text{X}^{(i)}$$

kx1                    nxk                    nx1                    nx1

Z就是对应降维之后的数据

实现代码：

```
# 映射数据
def projectData(X_norm,U,K):
    Z = np.zeros((X_norm.shape[0],K))

    U_reduce = U[:,0:K]      # 取前K个
    Z = np.dot(X_norm,U_reduce)
    return Z
```

过程总结：

```
Sigma = X'*X/m
U,S,V = svd(Sigma)
Ureduce = U[:,0:k]
Z = Ureduce'*x
```

## 5、数据恢复

因为： $Z^{(i)} = U_{\text{reduce}}^T * X^{(i)}$ ,

所以： $X_{\text{approx}} = (U_{\text{reduce}}^T)^{-1} Z$  (注意这里是X的近似值)

又因为Ureduce为正定矩阵，【正定矩阵满足： $AA^T = A^TA = E$ ，所以： $A^{-1} = A^T$ 】，

所以这里： $X_{\text{approx}} = (U_{\text{reduce}}^{-1})^{-1} Z = U_{\text{reduce}} Z$

实现代码：

```
# 恢复数据
def recoverData(Z,U,K):
    X_rec = np.zeros((Z.shape[0],U.shape[0]))
    U_recude = U[:,0:K]
    X_rec = np.dot(Z,np.transpose(U_recude)) # 还原数据(近似)
    return X_rec
```

## 6、主成分个数的选择(即要降的维度)

如何选择

$$\text{投影误差 (project error)} : \frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{approx}^{(i)}\|^2$$

$$\text{总变差 (total variation)} : \frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2$$

$$\frac{\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2} \leq 0.01$$

若误差率 (error ratio) : , 则称99%保留差异性

误差率一般取1%, 5%, 10%等

如何实现

若是一个个试的话代价太大

之前 $U, S, V = svd(\Sigma)$ , 我们得到了 $S$ , 这里误差率error ratio:

$$\text{error ratio} = 1 - \frac{\sum_{i=1}^k S_{ii}}{\sum_{i=1}^n S_{ii}} \leq \text{threshold}$$

可以一点点增加K尝试。

## 7、使用建议

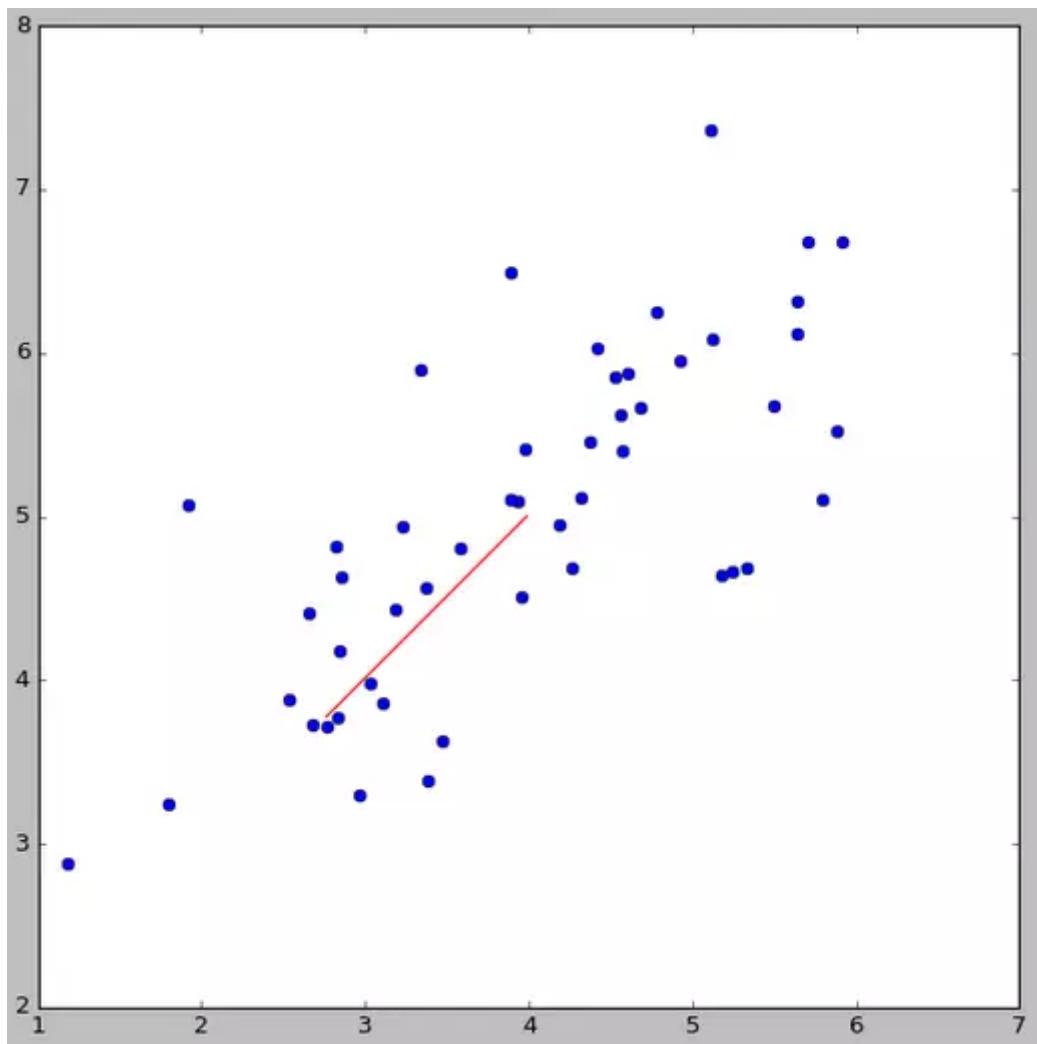
不要使用PCA去解决过拟合问题Overfitting，还是使用正则化的方法（如果保留了很高的差异性还是可以的）

只有在原数据上有好的结果，但是运行很慢，才考虑使用PCA

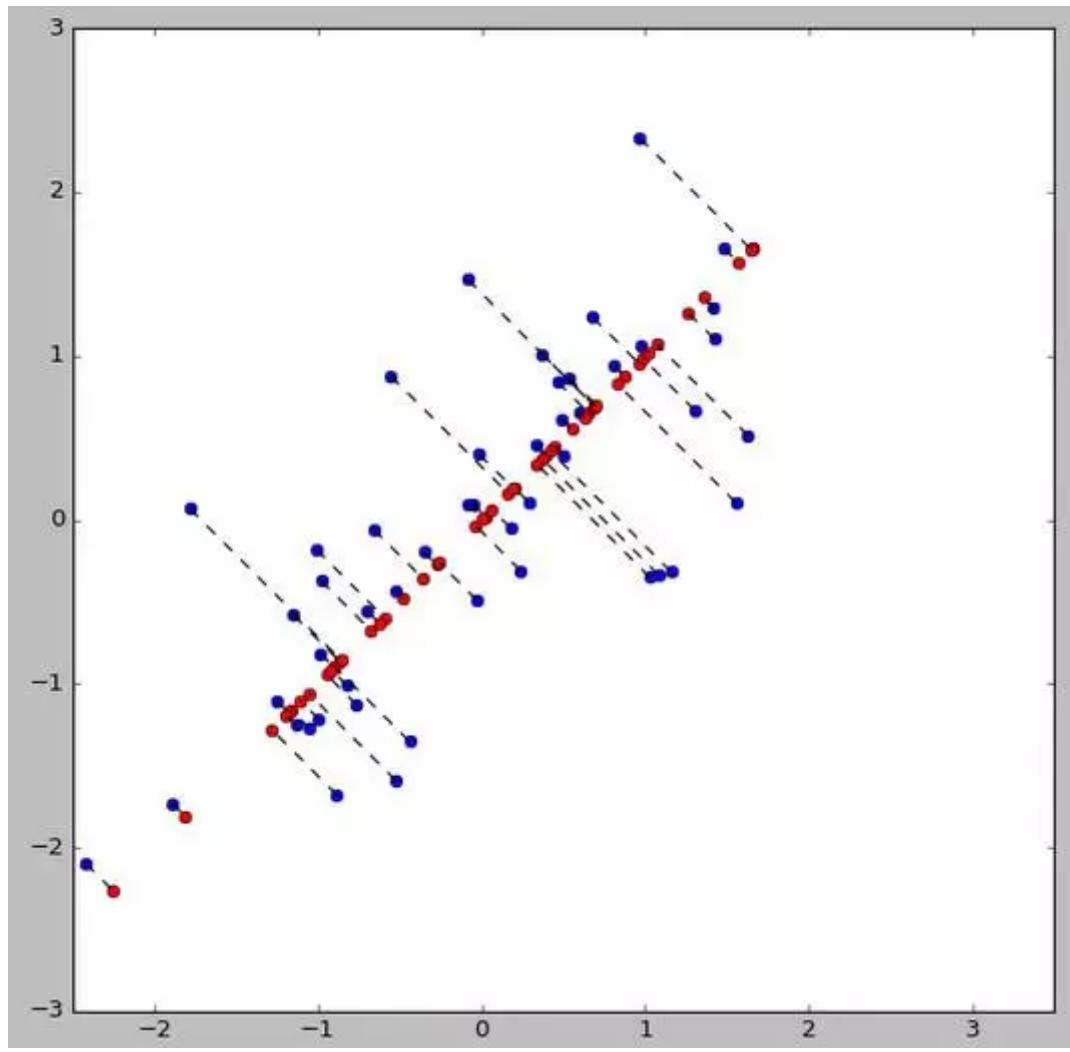
## 8、运行结果

2维数据降为1维

要投影的方向



2D降为1D及对应关系



人脸数据降维

原始数据



可视化部分U矩阵信息



恢复数据



## 9、使用scikit-learn库中的PCA实现降维

[https://github.com/lawlite19/MachineLearning\\_Python/blob/master/PCA/PCA.py\\_scikit-learn.py](https://github.com/lawlite19/MachineLearning_Python/blob/master/PCA/PCA.py_scikit-learn.py)

导入需要的包：

```
#-*- coding: utf-8 -*-
# Author: bob
# Date:2016.12.22
import numpy as np
from matplotlib import pyplot as plt
from scipy import io as spio
from sklearn.decomposition import pca
from sklearn.preprocessing import StandardScaler
```

## 归一化数据

```
'''归一化数据并作图'''

scaler = StandardScaler()
scaler.fit(X)
x_train = scaler.transform(X)
```

使用PCA模型拟合数据，并降维

n\_components对应要将的维度

```
'''拟合数据'''

K=1 # 要降的维度
model = pca.PCA(n_components=K).fit(x_train) # 拟合数据，n_components定义要降的维度
Z = model.transform(x_train) # transform就会执行降维操作
```

数据恢复

model.components\_会得到降维使用的U矩阵

```
'''数据恢复并作图'''

Ureduce = model.components_ # 得到降维用的Ureduce
x_rec = np.dot(Z,Ureduce) # 数据恢复
```

## 七、异常检测 Anomaly Detection

全部代码

[https://github.com/lawlite19/MachineLearning\\_Python/blob/master/AnomalyDetection/AnomalyDetection.py](https://github.com/lawlite19/MachineLearning_Python/blob/master/AnomalyDetection/AnomalyDetection.py)

### 1、高斯分布（正态分布） Gaussian distribution

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

分布函数：

其中， $\mu$ 为数据的均值， $\sigma$ 为数据的标准差

$\sigma$ 越小，对应的图像越尖

## 参数估计 ( parameter estimation )

$$u = \frac{1}{m} \sum_{i=1}^m x^{(i)}$$

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - u)^2$$

## 2、异常检测算法

例子

训练集 :  $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ , 其中  $x \in R^n$

假设  $x_1, x_2, \dots, x_n$

相互独立 , 建立model模型 :

$$p(x) = p(x_1; u_1, \sigma_1^2) p(x_2; u_2, \sigma_2^2) \cdots p(x_n; u_n, \sigma_n^2) = \prod_{j=1}^n p(x_j; u_j, \sigma_j^2)$$

过程

选择具有代表异常的feature:xi

参数估计 :  $u_1, u_2, \dots, u_n; \sigma_1^2, \sigma_2^2, \dots, \sigma_n^2$

计算  $p(x)$ , 若是  $P(x) < \epsilon$  则认为异常 , 其中  $\epsilon$  为我们要求的概率的临界值 threshold

这里只是单元高斯分布 , 假设了 feature 之间是独立的 , 下面会讲到多元高斯分布 , 会自动捕捉到 feature 之间的关系

参数估计实现代码

```
# 参数估计函数 ( 就是求均值和方差 )
def estimateGaussian(X):
    m,n = X.shape
```

```

mu = np.zeros((n,1))
sigma2 = np.zeros((n,1))

mu = np.mean(X, axis=0) # axis=0表示列，每列的均值
sigma2 = np.var(X, axis=0) # 求每列的方差
return mu,sigma2

```

### 3、评价p(x)的好坏，以及ε的选取

对偏斜数据的错误度量

因为数据可能是非常偏斜的（就是 $y=1$ 的个数非常少，( $y=1$ 表示异常)），所以可以使用Precision/Recall，计算F1Score(在CV交叉验证集上)

例如：预测癌症，假设模型可以得到99%能够预测正确，1%的错误率，但是实际癌症的概率很小，只有0.5%，那么我们始终预测没有癌症 $y=0$ 反而可以得到更小的错误率。使用error rate来评估就不科学了。

如下图记录：

		真实类	
		1	0
预测类	1	True Positive	False Positive
	0	False Negative	True Negative

$$\text{Precision} = \frac{TP}{TP + FP} \text{, 即：正确预测正样本/所有预测正样本}$$

$$\text{Recall} = \frac{TP}{TP + FN} \text{, 即：正确预测正样本/真实值为正样本}$$

总是让 $y=1$ (较少的类)，计算Precision和Recall

$$F_1 Score = 2 \frac{PR}{P+R}$$

还是以癌症预测为例，假设预测都是no-cancer，TN=199，FN=1，TP=0，FP=0，所以：  
Precision=0/0，Recall=0/1=0，尽管accuracy=199/200=99.5%，但是不可信。

$\epsilon$ 的选取

尝试多个 $\epsilon$ 值，使F1Score的值高

实现代码

```
# 选择最优的epsilon，即：使F1Score最大
def selectThreshold(yval,pval):
    """初始化所需变量"""
    bestEpsilon = 0.
    bestF1 = 0.
    F1 = 0.

    step = (np.max(pval)-np.min(pval))/1000
    """计算"""
    for epsilon in np.arange(np.min(pval),np.max(pval),step):
        cvPrecision = pval<epsilon
        tp = np.sum((cvPrecision == 1) & (yval == 1)).astype(float) # sum求和是int型的，需要转为float
        fp = np.sum((cvPrecision == 1) & (yval == 0)).astype(float)
        fn = np.sum((cvPrecision == 1) & (yval == 0)).astype(float)
        precision = tp/(tp+fp) # 精准度
        recall = tp/(tp+fn) # 召回率
        F1 = (2*precision*recall)/(precision+recall) # F1Score计算公式
        if F1 > bestF1: # 修改最优的F1 Score
            bestF1 = F1
            bestEpsilon = epsilon
    return bestEpsilon,bestF1
```

#### 4、选择使用什么样的feature（单元高斯分布）

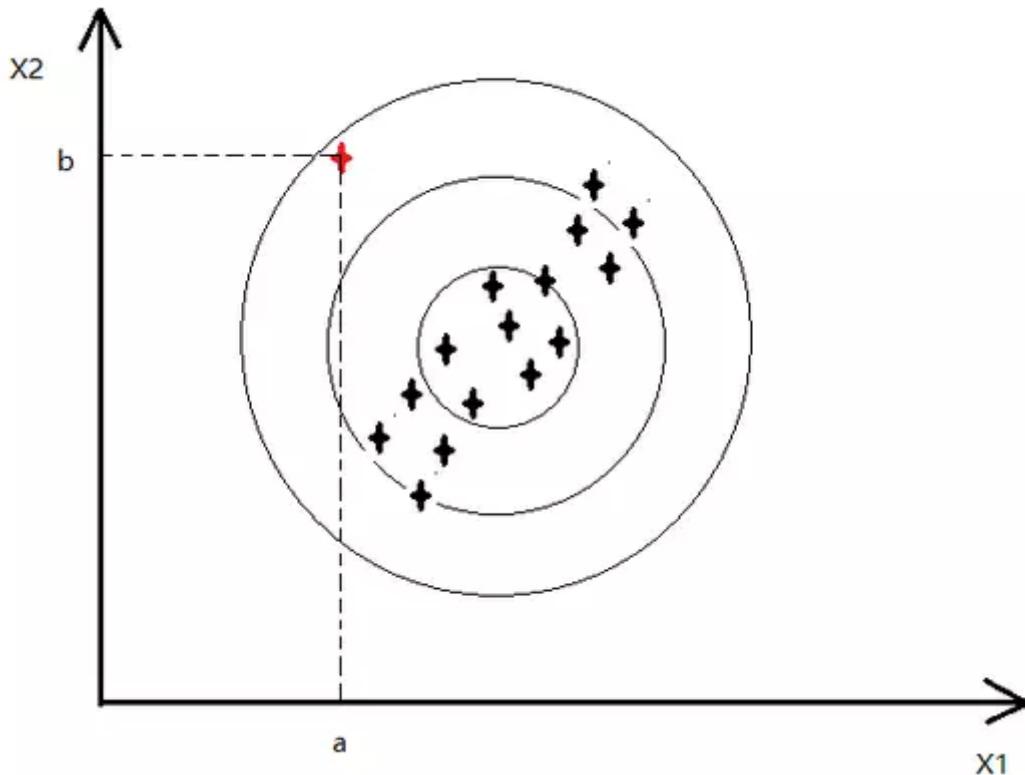
如果一些数据不是满足高斯分布的，可以变化一下数据，例如 $\log(x+C)$ , $x^{(1/2)}$ 等

如果 $p(x)$ 的值无论异常与否都很大，可以尝试组合多个feature,(因为feature之间可能是有关系的)

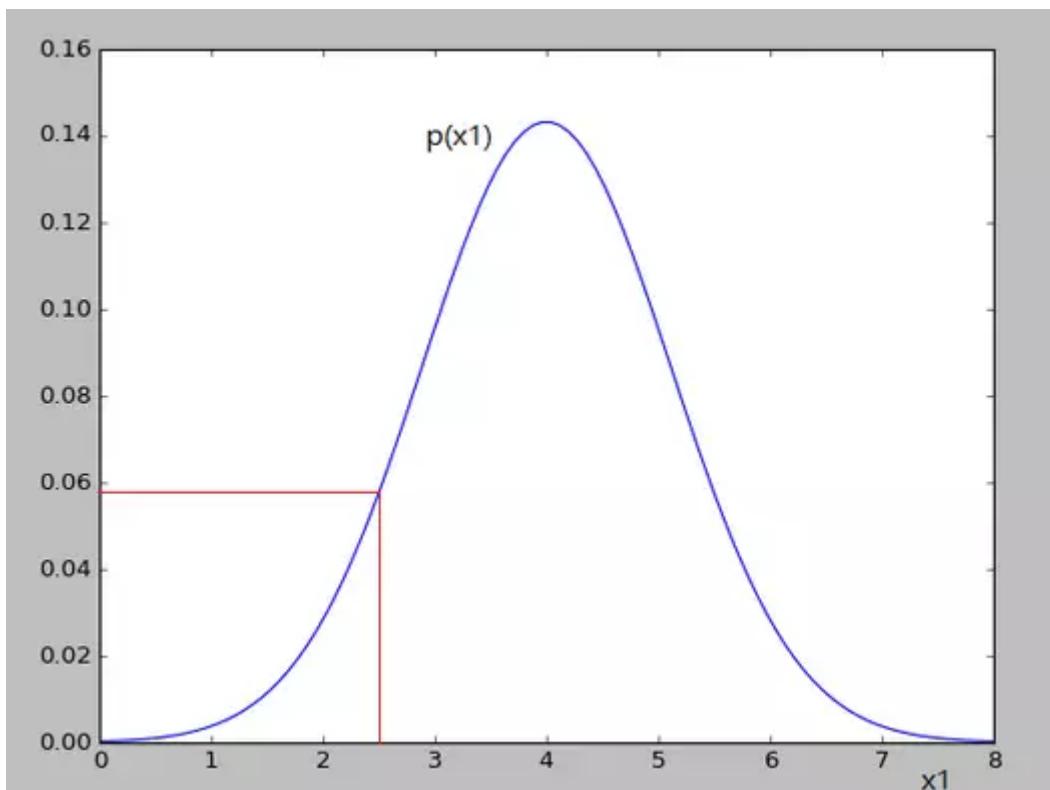
## 5、多元高斯分布

单元高斯分布存在的问题

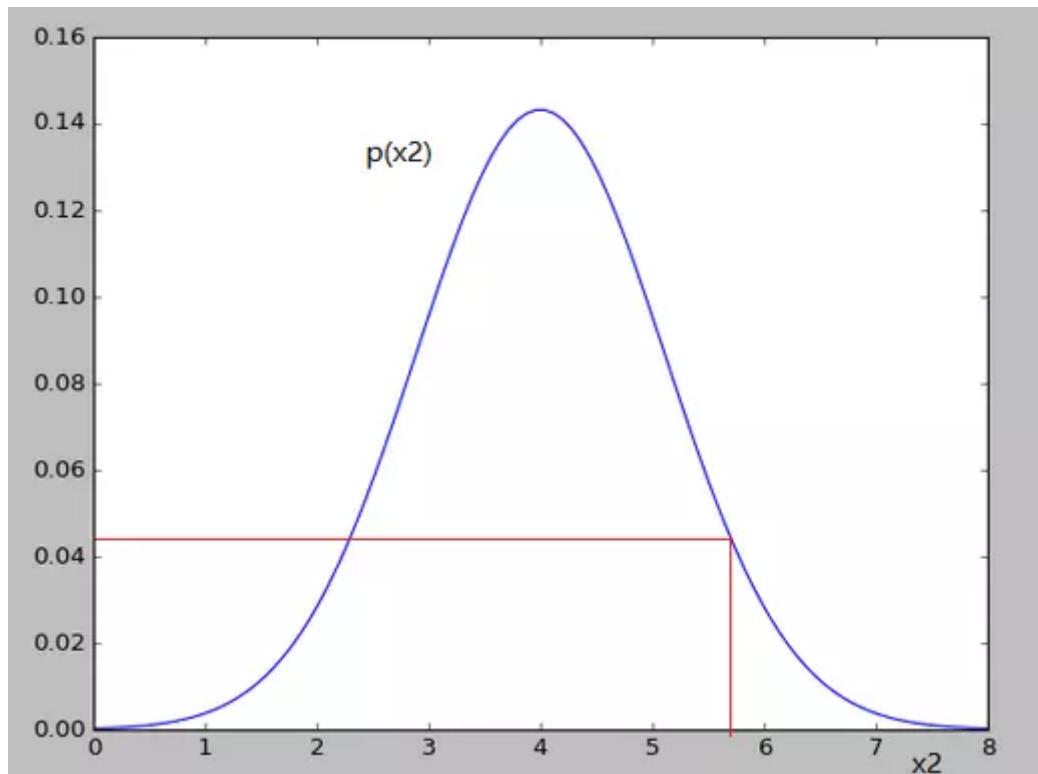
如下图，红色的点为异常点，其他的都是正常点（比如CPU和memory的变化）



$x_1$ 对应的高斯分布如下：



$x_2$ 对应的高斯分布如下：



可以看出对应的 $p(x_1)$ 和 $p(x_2)$ 的值变化并不大，就不会认为异常

因为我们认为feature之间是相互独立的，所以如上图是以正圆的方式扩展

### 多元高斯分布

$x \in R^n$ ，并不是建立 $p(x_1), p(x_2) \dots p(x_n)$ ，而是统一建立 $p(x)$

其中参数： $\mu \in R^n, \Sigma \in R^{n \times n}$ ,  $\Sigma$ 为协方差矩阵

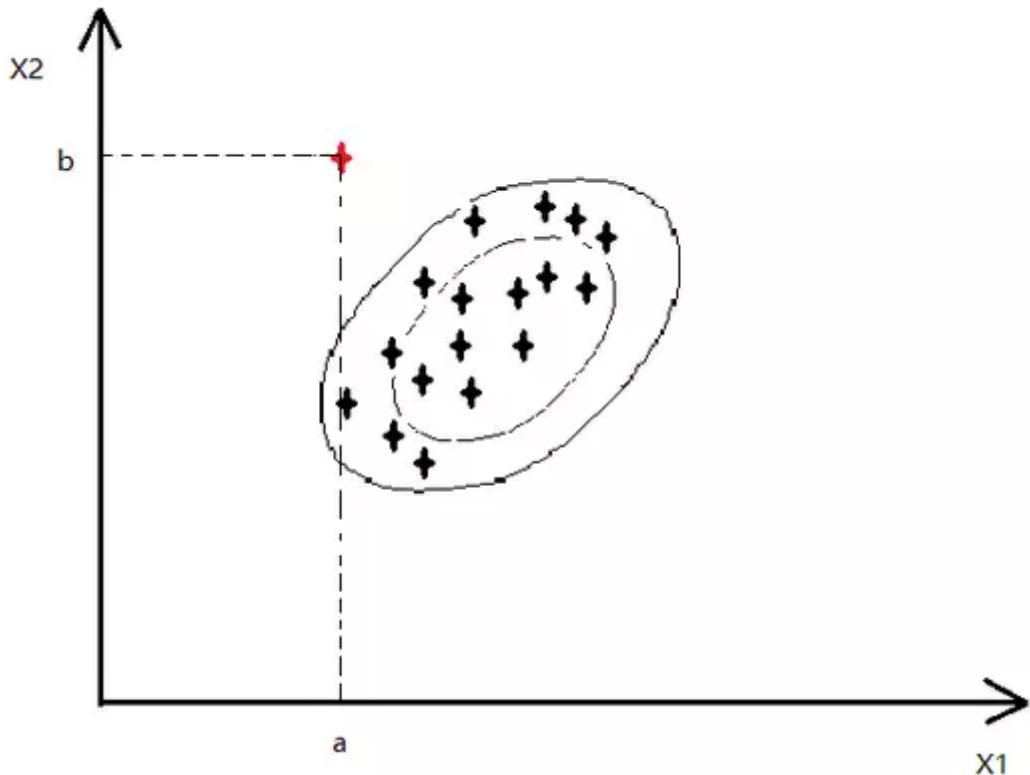
$$p(x) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)}$$

同样， $|\Sigma|$ 越小， $p(x)$ 越尖

例如：

$$\Sigma = \begin{bmatrix} 1 & 0.3 \\ 0.3 & 1 \end{bmatrix}$$

表示 $x_1, x_2$ 正相关，即 $x_1$ 越大， $x_2$ 也就越大，如下图，也就可以将红色的异常点检查出了



若：

$$\Sigma = \begin{bmatrix} 1 & -0.3 \\ -0.3 & 1 \end{bmatrix}$$

表示 $x_1, x_2$ 负相关

实现代码：

```
# 多元高斯分布函数
def multivariateGaussian(X,mu,Sigma2):
    k = len(mu)
    if (Sigma2.shape[0]>1):
```

```
Sigma2 = np.diag(Sigma2)
'''多元高斯分布函数'''
X = X-mu
argu = (2*np.pi)**(-k/2)*np.linalg.det(Sigma2)**(-0.5)
p = argu*np.exp(-0.5*np.sum(np.dot(X,np.linalg.inv(Sigma2))*X, axis=1)) # axis表示每行
return p
```

## 6、单元和多元高斯分布特点

单元高斯分布

人为可以捕捉到feature之间的关系时可以使用

计算量小

多元高斯分布

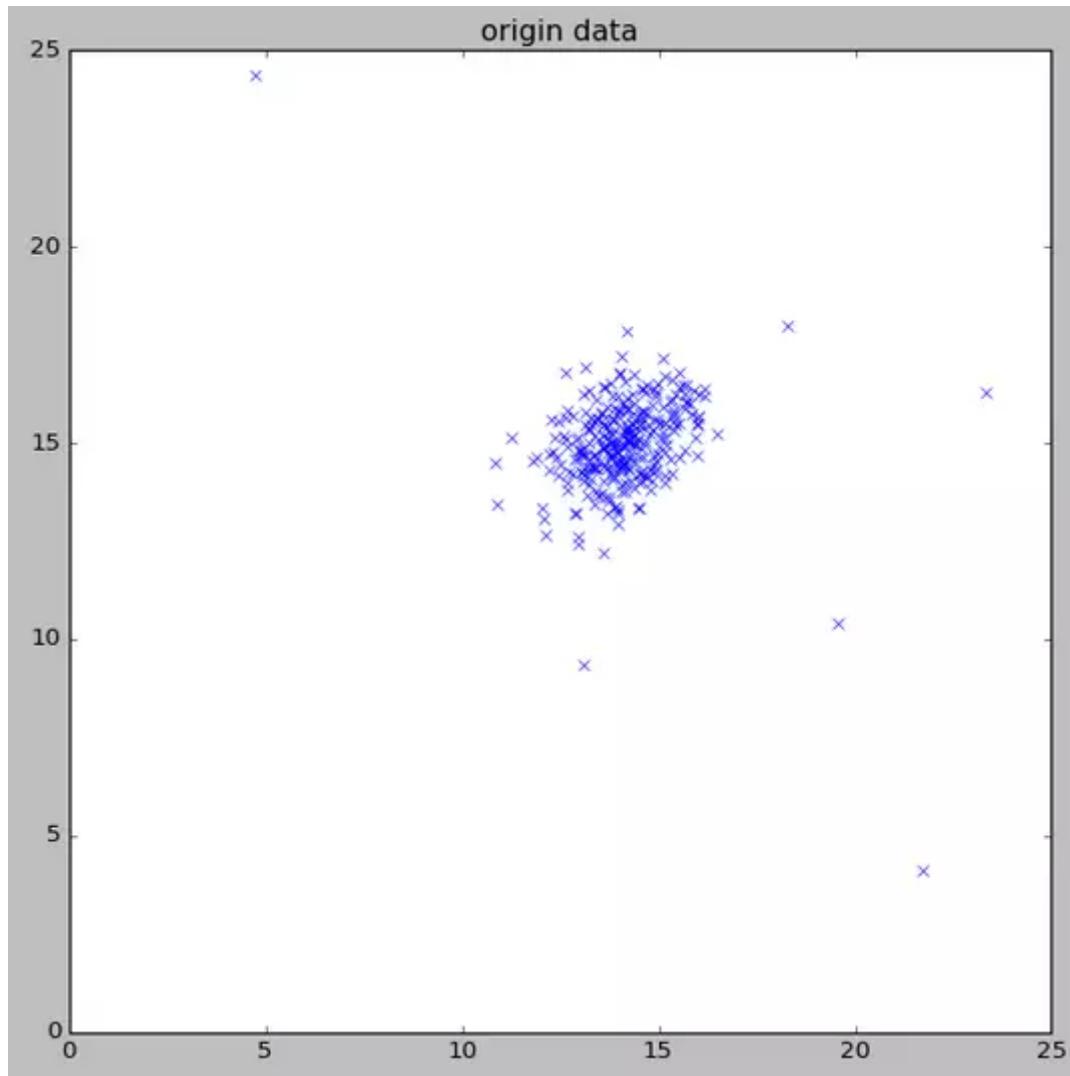
自动捕捉到相关的feature

计算量大，因为：

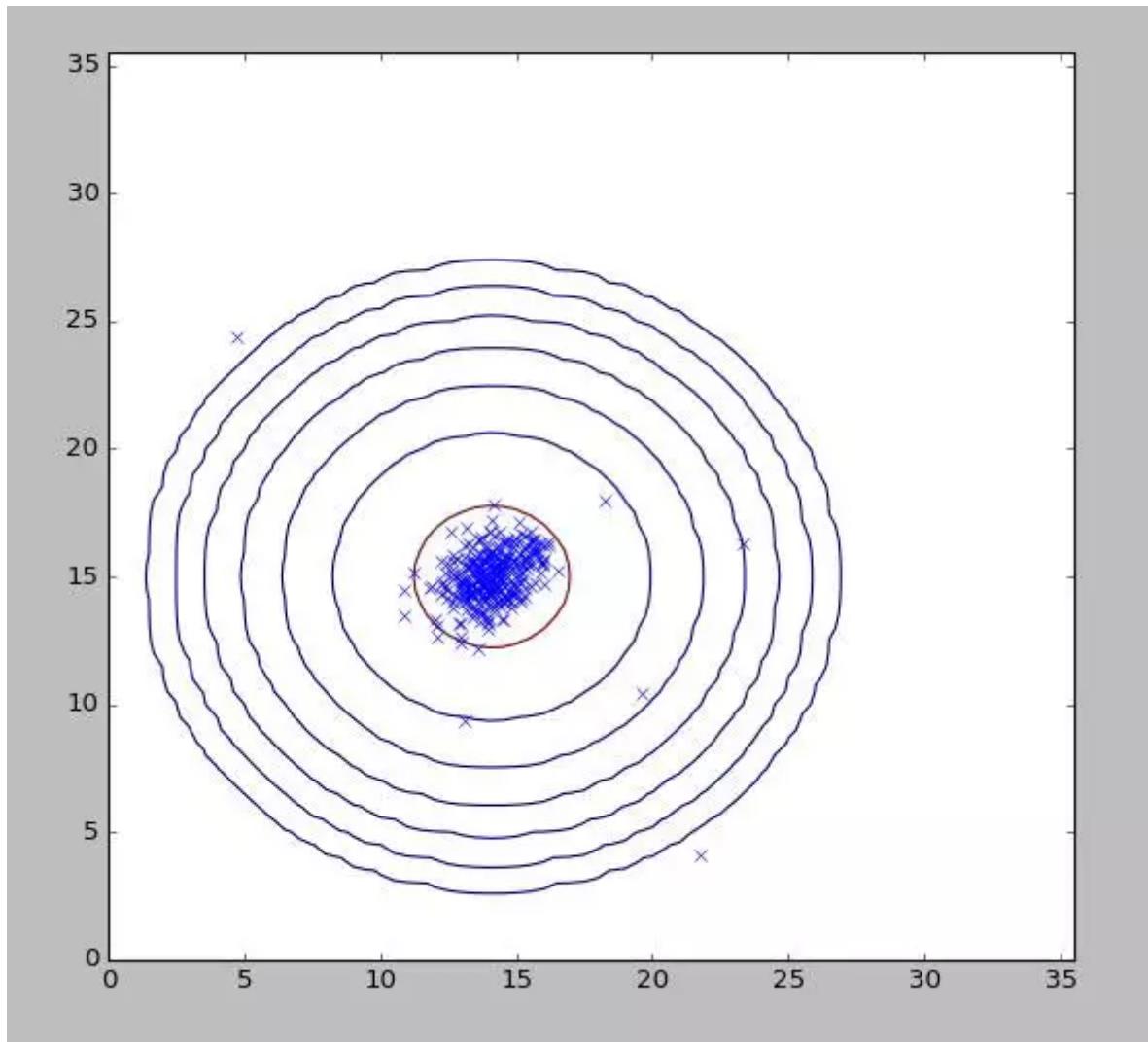
$m > n$  或  $\Sigma$  可逆时可以使用。（若不可逆，可能有冗余的  $x$ ，因为线性相关，不可逆，或者就是  $m < n$ ）

## 7、程序运行结果

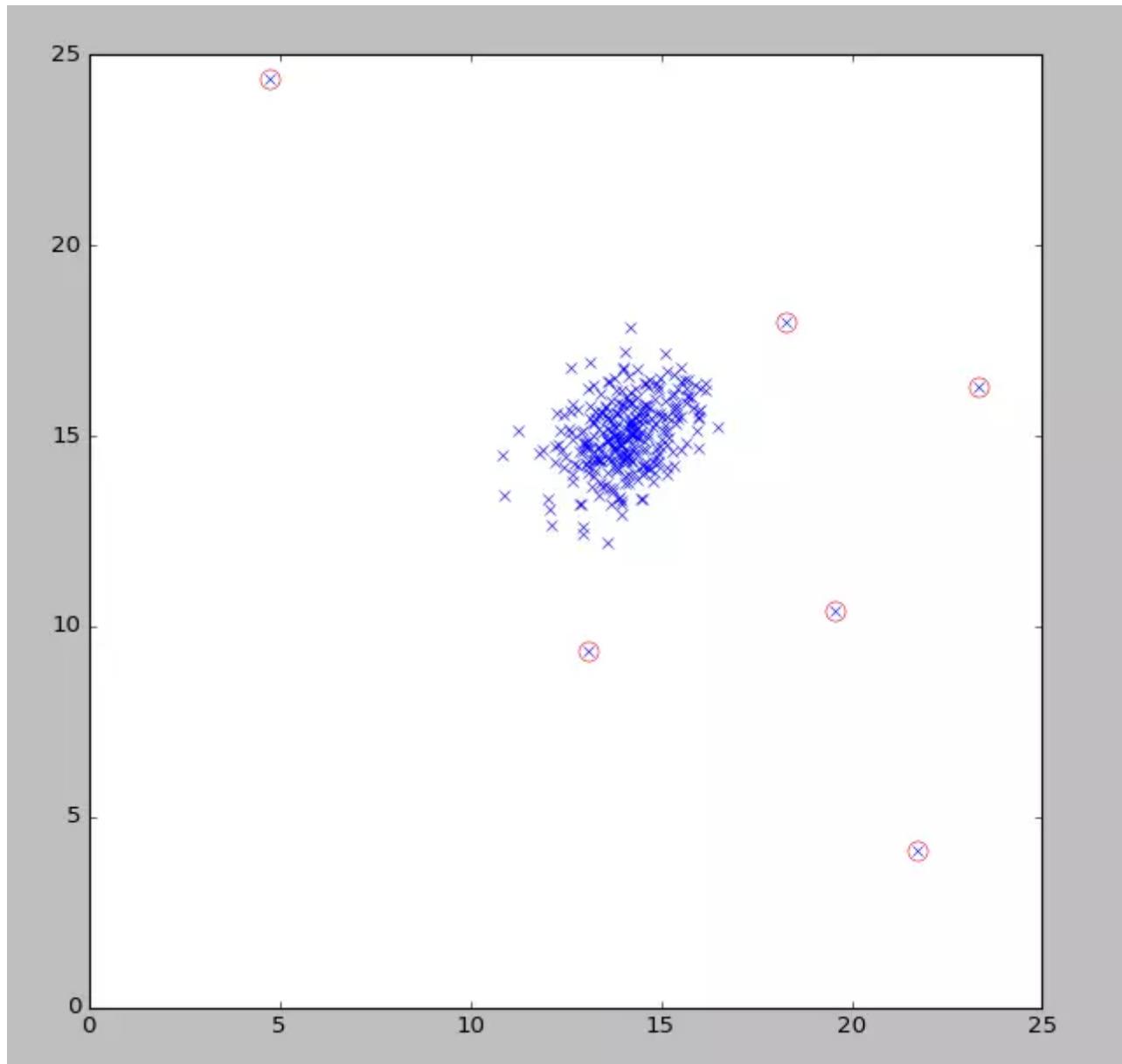
显示数据



等高线



异常点标注



原文地址

[https://github.com/lawlite19/MachineLearning\\_Python#](https://github.com/lawlite19/MachineLearning_Python#)

---

### 猜你喜欢

---

经验 | 如何高效学Python ?

一文总结学习Python的14张思维导图

如何在Python中用LSTM网络进行时间序列预测

疯狂上涨的 Python , 开发者应从 2.x 还是 3.x 着手 ?

2017年首份中美数据科学对比报告 , Python受欢迎度排名第一 , 美国数据工作者年薪中位数高达 11万美金



[点击阅读原文，查看文中所涉及的代码](#)

[阅读原文](#)