# 机器学习算法Python实现

[MIT license](https://github.com/lawlite19/MachineLearning_Python/blob/master/LICENSE)

## 目录

* [机器学习算法Python实现](#机器学习算法python实现)
  + [一、线性回归](#一线性回归)
    - [1、代价函数](#1代价函数)
    - [2、梯度下降算法](#2梯度下降算法)
    - [3、均值归一化](#3均值归一化)
    - [4、最终运行结果](#4最终运行结果)
    - [5、使用scikit-learn库中的线性模型实现](#5使用scikit-learn库中的线性模型实现)
  + [二、逻辑回归](#二逻辑回归)
    - [1、代价函数](#1代价函数)
    - [2、梯度](#2梯度)
    - [3、正则化](#3正则化)
    - [4、S型函数（即）](#4s型函数即)
    - [5、映射为多项式](#5映射为多项式)
    - [6、使用的优化方法](#6使用scipy的优化方法)
    - [7、运行结果](#7运行结果)
    - [8、使用scikit-learn库中的逻辑回归模型实现](#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](#逻辑回归_手写数字识别_onevsall)
    - [1、随机显示100个数字](#1随机显示100个数字)
    - [2、OneVsAll](#2onevsall)
    - [3、手写数字识别](#3手写数字识别)
    - [4、预测](#4预测)
    - [5、运行结果](#5运行结果)
    - [6、使用scikit-learn库中的逻辑回归模型实现](#6使用scikit-learn库中的逻辑回归模型实现)
  + [三、BP神经网络](#三bp神经网络)
    - [1、神经网络model](#1神经网络model)
    - [2、代价函数](#2代价函数)
    - [3、正则化](#3正则化)
    - [4、反向传播BP](#4反向传播bp)
    - [5、BP可以求梯度的原因](#5bp可以求梯度的原因)
    - [6、梯度检查](#6梯度检查)
    - [7、权重的随机初始化](#7权重的随机初始化)
    - [8、预测](#8预测)
    - [9、输出结果](#9输出结果)
  + [四、SVM支持向量机](#四svm支持向量机)
    - [1、代价函数](#1代价函数)
    - [2、Large Margin](#2large-margin)
    - [3、SVM Kernel（核函数）](#3svm-kernel核函数)
    - [4、使用中的模型代码](#4使用scikit-learn中的svm模型代码)
    - [5、运行结果](#5运行结果)
  + [五、K-Means聚类算法](#五k-means聚类算法)
    - [1、聚类过程](#1聚类过程)
    - [2、目标函数](#2目标函数)
    - [3、聚类中心的选择](#3聚类中心的选择)
    - [4、聚类个数K的选择](#4聚类个数k的选择)
    - [5、应用——图片压缩](#5应用图片压缩)
    - [6、使用scikit-learn库中的线性模型实现聚类](#6使用scikit-learn库中的线性模型实现聚类)
    - [7、运行结果](#7运行结果)
  + [六、PCA主成分分析（降维）](#六pca主成分分析降维)
    - [1、用处](#1用处)
    - [2、2D-->1D，nD-->kD](#22d--1dnd--kd)
    - [3、主成分分析PCA与线性回归的区别](#3主成分分析pca与线性回归的区别)
    - [4、PCA降维过程](#4pca降维过程)
    - [5、数据恢复](#5数据恢复)
    - [6、主成分个数的选择（即要降的维度）](#6主成分个数的选择即要降的维度)
    - [7、使用建议](#7使用建议)
    - [8、运行结果](#8运行结果)
    - [9、使用scikit-learn库中的PCA实现降维](#9使用scikit-learn库中的pca实现降维)
  + [七、异常检测 Anomaly Detection](#七异常检测-anomaly-detection)
    - [1、高斯分布（正态分布）](#1高斯分布正态分布gaussian-distribution)
    - [2、异常检测算法](#2异常检测算法)
    - [3、评价的好坏，以及的选取](#3评价px的好坏以及ε的选取)
    - [4、选择使用什么样的feature（单元高斯分布）](#4选择使用什么样的feature单元高斯分布)
    - [5、多元高斯分布](#5多元高斯分布)
    - [6、单元和多元高斯分布特点](#6单元和多元高斯分布特点)
    - [7、程序运行结果](#7程序运行结果)

## 一、[线性回归](file:///C:\LinearRegression)

* [全部代码](C:\\LinearRegression\\LinearRegression.py)

### 1、代价函数

* 
* 其中： 
* 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

* 代价函数对求偏导得到：  
  
* 所以对theta的更新可以写为：  
  
* 其中为学习速率，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+△x)=f(x)+f'(x)\*△x+o(△x)
* 令：△x=-α\*f'(x) ,即负梯度方向乘以一个很小的步长α
* 将△x代入泰勒展开式中：f(x+△x)=f(x)-α\*[f'(x)]²+o(△x)
* 可以看出，α是取得很小的正数，[f'(x)]²也是正数，所以可以得出：f(x+△x)<=f(x)
* 所以沿着**负梯度**放下，函数在减小，多维情况一样。
* 实现代码
* # 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  为所有此feture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  
  enter description here

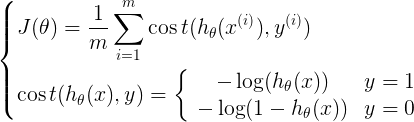
### 5、[使用scikit-learn库中的线性模型实现](file:///C:\LinearRegression\LinearRegression_scikit-learn.py)

* 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](file:///C:\LogisticRegression)

* [全部代码](C:\\LogisticRegression\\LogisticRegression.py)

### 1、代价函数

* 
* 可以综合起来为：  其中： 
* 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
* 的图像如下，即y=1时： enter description here

可以看出，当趋于1，y=1,与预测值一致，此时付出的代价cost趋于0，若趋于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 - 同理的图像如下（y=0）：  
enter description here

### 2、梯度

* 同样对代价函数求偏导：   
  可以看出与线性回归的偏导数一致
* 推到过程 enter description here

### 3、正则化

* 目的是为了防止过拟合
* 在代价函数中加上一项
* 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的长度一置  
    
  h = 1.0/(1.0+np.exp(-z))  
  return h

### 5、映射为多项式

* 因为数据的feture可能很少，导致偏差大，所以创造出一些feture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是其余测参数，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  
  enter description here enter description here
* data2决策边界和准确度  
  enter description here enter description here

### 8、[使用scikit-learn库中的逻辑回归模型实现](file:///C:\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()  
   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](C:\\LogisticRegression)

* [全部代码](C:\\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 enter description here 灰度图： enter description here
* 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，这样有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... enter description here
* 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....** enter description here
* 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：  
  enter description here

### 6、[使用scikit-learn库中的逻辑回归模型实现](file:///C:\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） enter description here

## 三、BP神经网络

* [全部代码](C:\\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 enter description here
* 所以可以得到：
* 隐含层：  
    
    
  
* 输出层  
   其中，**S型函数**，也成为**激励函数**
* 可以看出 为3x4的矩阵，为1x4的矩阵
*  ==》j+1的单元数x（j层的单元数+1）

### 2、代价函数

* 假设最后输出的，即代表输出层有K个单元
*  其中，代表第i个单元输出
* 与逻辑回归的代价函数差不多，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
* -->第l层unit的个数
* 正则化后的**代价函数**为  
  enter description here
* 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(θ),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,记为-->l层第j个单元的误差
* 《===》（向量化）
* 
* 
* 没有，因为对于输入没有误差
* 因为S型函数的导数为：，所以上面的和可以在前向传播中计算出来
* 反向传播计算梯度的过程为：
* （是大写的）
* for i=1-m:  
   -  
  -正向传播计算（l=2,3,4...L）  
  -反向计算、...；  
  -  
  -  
  
* 最后，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).copy() # 这里使用copy函数，否则下面修改Theta的值，nn\_params也会一起修改  
   Theta2 = nn\_params[hidden\_layer\_size\*(input\_layer\_size+1):length].reshape(num\_labels,hidden\_layer\_size+1).copy()  
   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  
    
    
  '''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,:])\*sigmoidGradient(z3[i,:]) # 均方误差的误差率  
   delta3[i,:] = h[i,:]-class\_y[i,:] # 交叉熵误差率  
   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))  
    
  Theta1[:,0] = 0  
  Theta2[:,0] = 0   
  '''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。 enter description here
* 求误差更详细的推导过程： enter description here

### 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：  
  enter description here
* 随机显示100个手写数字  
  enter description here
* 显示theta1权重  
  enter description here
* 训练集预测准确度  
  enter description here
* 归一化后训练集预测准确度  
  enter description here

## 四、SVM支持向量机

### 1、代价函数

* 在逻辑回归中，我们的代价为：  
  ，  
  其中：，
* 如图所示，如果y=1，cost代价函数如图所示  
  enter description here  
  我们想让，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**的函数代替逻辑回归中的cost
* 当y=0时同样，用代替 enter description here
* 最终得到的代价函数为：  
    
  最后我们想要
* 之前我们逻辑回归中的代价函数为：  
    
  可以认为这里的，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  
  enter description here
* 先说一下向量内积
* ，
* 表示u的**欧几里得范数**（欧式范数），
* 向量V在向量u上的投影的长度记为p，则：向量内积：  
     
   enter description here  
  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(θ),当margin最大时，C的乘积项要很小，所以近似为：  
  ，  
  我们最后的目的就是求使代价最小的θ
* 由  
  可以得到：  
  ，p即为x在θ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θ上的投影为p,则或者，若是p很小，则需要很大，这与我们要求的θ使最小相违背，**所以**最后求的是large margin  
  enter description here

### 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**
* 高斯核函数为：  
  假设如图几个点， enter description here 令：  
    
   . . .
* 可以看出，若是x与距离较近，==》，（即相似度较大）  
  若是x与距离较远，==》，（即相似度较低）
* 高斯核函数的σ越小，f下降的越快  
  enter description here enter description here
* 如何选择初始的
* 训练集：
* 选择：
* 对于给出的x，计算f,令：所以：
* 最小化J求出θ，  
   
* 如果，==》预测y=1

### 4、使用scikit-learn中的SVM模型代码

* [全部代码](C:\\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：  
  enter description here
* 线性不可分的决策边界：  
  enter description here

## 五、K-Means聚类算法

* [全部代码](C:\\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  
   enter description here
* 重新计算聚类中心，移动一次  
   enter description here
* 最后10步之后的聚类中心  
   enter description here
* 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**
* 
* 最后我们想得到：  
  enter description here
* 其中表示第i条数据距离哪个类中心最近，
* 其中即为聚类的中心

### 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 enter description here
* 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](file:///C:\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  
  enter description here
* 图片压缩  
  enter description here

## 六、PCA主成分分析（降维）

* [全部代码](C:\\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 enter description here
* 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-->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

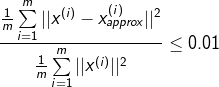
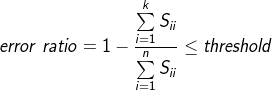
### 4、PCA降维过程

* 数据预处理（均值归一化）
* 公式：
* 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](file:///C:\Users\eyunxli\Downloads\images\LinearRegression_01.png) 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Σ（Covariance Matrix）：
* 注意这里的Σ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nxn,n为feature的维度
* 实现代码： Sigma = np.dot(np.transpose(X\_norm),X\_norm)/m # 求Sigma
* 计算Σ的特征值和特征向量
* 可以是用svd奇异值分解函数：U,S,V = svd(Σ)
* 返回的是与Σ同样大小的对角阵S（由Σ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
* enter description here
* **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）
* enter description here
* 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：
* 所以： （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：，所以：】，所以这里：
* 
* 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
* **投影误差**（project error）：
* **总变差**（total variation）:
* 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:  
   
* 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  
  enter description here
* 2D降为1D及对应关系  
  enter description here
* 人脸数据降维
* 原始数据  
   enter description here
* 可视化部分U矩阵信息  
   enter description here
* 恢复数据  
   enter description here

### 9、[使用scikit-learn库中的PCA实现降维](file:///C:\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](C:\\AnomalyDetection\\AnomalyDetection.py)

### 1、高斯分布（正态分布）Gaussian distribution

* 分布函数：
* 其中，u为数据的**均值**，σ为数据的**标准差**
* σ越**小**，对应的图像越**尖**
* 参数估计（parameter estimation）
* 
* 

$${\sigma ^2} = {1 \over m}\sum\limits\_{i = 1}^m {{{({x^{(i)}} - u)}^2}} $$

### 2、异常检测算法

* 例子
* 训练集：,其中
* 假设相互独立，建立model模型：
* 过程
* 选择具有代表异常的feature:xi
* 参数估计：
* 计算p(x),若是P(x)<ε则认为异常，其中ε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：  
   enter description here
*  ，即：**正确预测正样本/所有预测正样本**
*  ，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
* 
* 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。
* ε的选取
* 尝试多个ε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).ravel()).astype(float) # sum求和是int型的，需要转为float  
   fp = np.sum((cvPrecision == 1) & (yval == 0).ravel()).astype(float)  
   fn = np.sum((cvPrecision == 0) & (yval == 1).ravel()).astype(float)  
   precision = tp/(tp+fp) # 精准度  
   recision = tp/(tp+fn) # 召回率  
   F1 = (2\*precision\*recision)/(precision+recision) # 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）  
   enter description here
* x1对应的高斯分布如下：  
   enter description here
* x2对应的高斯分布如下：  
   enter description here
* 可以看出对应的p(x1)和p(x2)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
* ，并不是建立p(x1),p(x2)...p(xn)，而是统一建立p(x)
* 其中参数：,Σ为**协方差矩阵**
* 
* 同样，|Σ|越小，p(x)越尖
* 例如：  
   enter description here，  
   表示x1,x2**正相关**，即x1越大，x2也就越大，如下图，也就可以将红色的异常点检查出了 enter description here  
   若：  
   enter description here，  
   表示x1,x2**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Σ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  
  enter description here
* 等高线  
  enter description here
* 异常点标注  
  enter description here