# 初识机器学习

## 监督学习

监督学习（Supervised Learning）,输入样本已被标记。有两类问题，回归(Regression)和分类（Classification）。

Regression的输出是连续的，Classification输出离散。

**典型案例：**

Regression根据已知楼盘房价推测新楼盘房价。

Classification根据肿瘤样本的良性/恶性，推测新肿瘤是良性还是恶性。

## 无监督学习

无监督学习（Unsupervised Learning），输入样本没有被标记，需要根据样本间的相似性对样本集分类。

**典型案例：**

用户细分

混合音频的剥离

# 线性回归

监督学习中的回归问题，分为单变量线性回归（Linear regression with one variable）和多变量线性回归（(Linear Regression with Multiple Variables）。

一般公式为：

其中，，为变量数量，也叫特征数量。

矩阵公式：

## 代价函数

1. **函数原型**

文本

描述已自动生成

1. **python实现**

|  |
| --- |
| #代价函数  #x, y, theta为矩阵变量  def costFunctionJ(x, y, theta) :  #计算(x\*theta - y) ^ 2  inner = np.power(((x.dot(theta.T)) - y), 2)  #计算  累加和/2m  return np.sum(inner) / (2 \* len(x)) |

1. **原理**

假定训练实例原本为97x1的矩阵，为了计算矩阵相乘，前面插入一列1，为97x2矩阵。

矩阵为每个实例的输出，也是97x1矩阵。

m为实例数量，97。

矩阵为2x1矩阵，即，其中 = 1，每次运算有两个值。

回到代价函数公式，其中：

，

换成矩阵形式，即矩阵和矩阵相乘，，所得结果是97x1矩阵，包含了m个训练实例的每一个计算结果。

矩阵运算包含了每一个训练实例的计算结果与实际结构的差值。

python代码np.power((x.dot(theta.T), 2)，对差值矩阵的每个元素求平方。numpy使用dot进行矩阵乘法，使用multiply进行矩阵点乘。不要使用\*，容易乱。

最后的np.sum(inner) / (2 \* len(x))，也就是公式中功能。

## 批量梯度下降

1. **函数原型**

文本

描述已自动生成

1. **python实现**

|  |
| --- |
| def gradientDescent(x, y, theta, alpha, iters):      #theta矩阵清零      temp = np.matrix(np.zeros(theta.shape))      #revel()将多维数组降为一维      #返回theta元素个数      parameters = int(theta.ravel().shape[1])      #比如iters=5,则cost为长度为5的数组，值为0      cost = np.zeros(iters)        for i in range(iters):          error = (x.dot(theta.T)) - y          print(error)            for j in range(parameters):              term = np.multiply(error, x[:,j])              temp[0,j] = theta[0,j] - ((alpha / len(x)) \* np.sum(term))            theta = temp          cost[i] = costFunctionJ(x, y, theta)        return theta, cost |

1. **原理**

对于，梯度下降公式：

其中：

公式展开：

其中：

n为的特征数，本例中n=2。

求导后：

其中，代表矩阵中第行、第列，也就是第个训练实例的第个特征。

对于：

设矩阵R为计算结果，实际上是矩阵R和矩阵X的第一列相乘矩阵的累加。

回到python：

代码error = (x.dot(theta.T)) – y即为。

代码term = np.multiply(error, x[:,j])，即为，其中为矩阵的第列。

代码np.sum(term)即为。

代码temp[0,j] = theta[0,j] - ((alpha / len(x)) \* np.sum(term))即为对的迭代过程，其中在这里的取值为0，1。

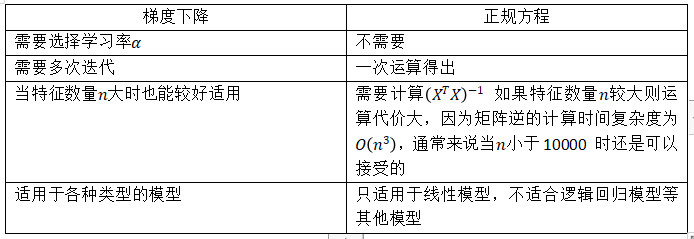
## 正规方程

正规方程（Normal Equation）是通过求解下面的方程来找出使得代价函数最小的参数的： 。

解得

TODO:推导过程涉及矩阵求导，目前不会，以后再推吧。

梯度下降与正规方程的比较：



总结，只要特征变量的数目并不大，标准方程是一个很好的计算参数的替代方法。具体地说，只要特征变量数量小于一万，我通常使用标准方程法，而不使用梯度下降法。

# 逻辑回归

Logistic Regression，监督学习中的分类问题，主要处理二分类中的线性可分问题。

对于线性可分，期望的假设函数处于[0, 1]范围，以便于判定：

当时，预测 = 1。

当时，预测 = 0。

因此结合了线性方程和Logistics/Sigmoid方程。

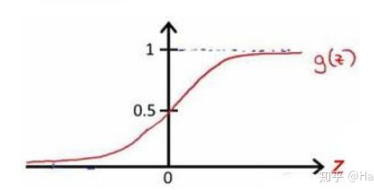
即：

的意义，给定输入变量,，计算输出变量等于1的可能性，也写作：

## 判定边界

Decision Boundary，就是判定的边界。

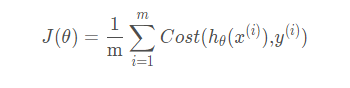
的图像为：



也就是确定后，的位置。

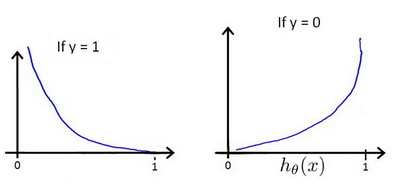
## 代价函数

逻辑回归的原始代价函数：



其中：

图像为：



这个函数的意义在于：

的情况，等于1时误差为0，越远离1，误差越大。

的情况，等于0时误差为0，越远离0，误差越大。

这样的函数特性符合实际预期，Cost二段式函数合并一下：

所以逻辑回归的最终代价函数为：

python代码实现也是根据这一条公式而来：

|  |
| --- |
| # Logistic Regression代价函数  # 返回值[jVal, gradient] - 为了适应fmin\_tcn调用  # 注意这个theta参数类型为数组  # 原因是fmin\_tnc函数自动调用代价函数时，  # 会对自动将theta转换成array\_like，因此参数theta的类型为array\_like  def costFunctionJ(theta, x, y):      #输入转换为矩阵形式      theta = np.matrix(theta)      x = np.matrix(x)      y = np.matrix(y)      #H(x)      hx= sigmoid(x.dot(theta.T))      first = np.multiply(-y, np.log(hx))      second = np.multiply((1 - y), np.log(1 - hx))      #矩阵形式，下面两个都可以，只是grad的形状不一样      #grad = 1.0/(len(X)) \* (hx - y).T.dot(x)      grad = 1.0/(len(X)) \* x.T.dot(hx - y)      return np.sum(first - second) / (len(X)), grad |

后续中， 这个代价函数会作为scipy库中的truncated newton (TNC) 算法的参数进行最优计算，为了适配TNC函数，代价函数的返回值参数表是[jVal, gradient]。

## 梯度下降

梯度下降公式：

代入后化简，得到的结果和线性方程的一致：

其中：

实际工程中基本不会使用这个算法，而是使用现成的算法库。

## TNC算法库

直接将代价函数、初始化值（[0, 0, 0]）、原始数据集传入fmin\_tnc函数进行调用。

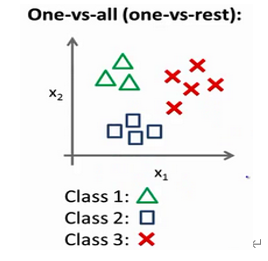
|  |
| --- |
| import scipy.optimize as opt  result = opt.fmin\_tnc(func = costFunctionJ, \  x0 = theta, args=(x, y), messages = 0) |

返回值：

|  |
| --- |
| 返回值[array, nfevel, rc]  @array : 最优theta数组  @nfevel: 函数评估的数量  @rc : 返回码  返回码rc定义：  -1 : Infeasible (lower bound > upper bound)  0 : Local minimum reached (|pg| ~= 0)  1 : Converged (|f\_n-f\_(n-1)| ~= 0)  2 : Converged (|x\_n-x\_(n-1)| ~= 0)  3 : Max. number of function evaluations reached  4 : Linear search failed  5 : All lower bounds are equal to the upper bounds  6 : Unable to progress  7 : User requested end of minimization |

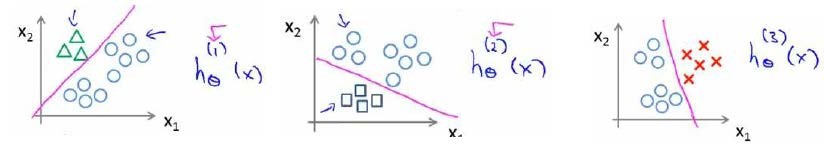
## 一对多问题

前面的例子只有两个取值的二元分类问题，如果是有多个取值，其图形是这样的：



其实可以转换思想，将“一对多”转换成“一对余”问题，上面图形的分类模型可以分为3个二元分类模型：

，其中



对每一个分类，将标记的一类记为1，其余标记都是0。分别计算、、。

预测时，给定，选择一个让 最大的，即，其中。

# 正则化

正则化用以解决过拟合问题。

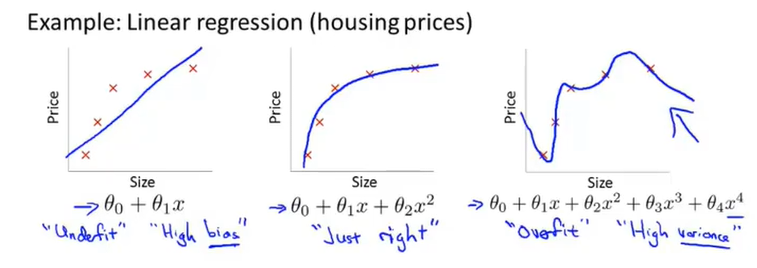
这里说的正则化指是L2正则化（还有L2正则）。

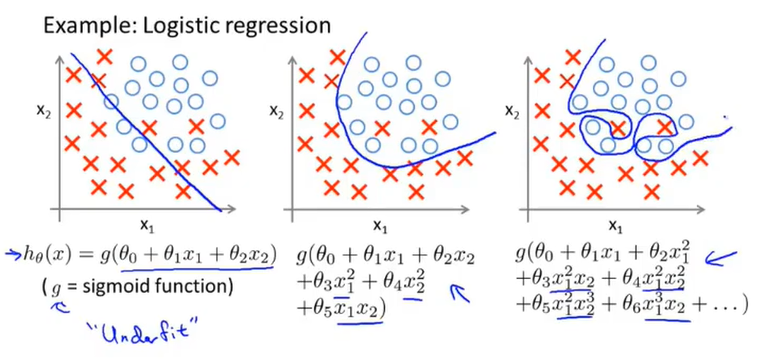
## 过拟合

为了满足现有的训练集，盲目加特征次数，搞的太复杂，结果泛化效果可能会不好。

（泛化，panalize，一个假设模型能够应用到新样本的能力）。

下面两组图，左一是欠拟合/高偏差，右一是过拟合/高方差。





解决办法：

1. 减少变量的个数：舍弃一些变量，保留更为重要的变量。但是，如果每个特征变量都对预测产生影响。当舍弃一部分变量时，也就舍弃了一些信息。所以，希望保留所有的变量。
2. 正则化：保留所有的变量，但是会减小特征变量的数量级（参数数值的大小θ(j)）。

## 线性回归正则化

线性回归正则化的代价函数就是在原来基础上加上正规项：

正规项是从1开始，即不进行处理。

### 梯度下降

线性回归正则化的梯度下降函数推导：

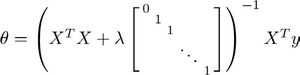
其中不处理：

其他的：

调整一下：

其中是一个略小于1的数，跟普通线性方程中的一样。

### 正规方程



其中矩阵的尺寸为。

这个公式还解决了普通线性回归正规方程的不可逆问题。

## 逻辑回归正则化

### 代价函数

代价函数中同样是加上正规项：

梯度下降不作处理：

之后的：

调整一下：

其中。

python代码，为了使用高级优化算法，同样要在代价函数里返回gradient：

|  |
| --- |
| #sigmoid函数  #参数可以是矩阵，或者实数  def sigmoid(z) :  return 1 / (1 + np.exp(-z))  def gradient(theta, x, y):  theta = np.matrix(theta)  x = np.matrix(x)  y = np.matrix(y)  hx= sigmoid(x.dot(theta.T))  return 1.0 / (len(x)) \* (x.T.dot(hx - y))  #这里要保证参数是矩阵，最好还是在函数内部进行一下处理  def gradientReg(theta, x, y, lam=1):  theta = np.matrix(theta)  x = np.matrix(x)  y = np.matrix(y)  #实数\*矩阵  reg=(lam/len(x))\*theta.T  reg[0] = 0 # 第一项没有惩罚因子  return gradient(theta, x, y) + reg  #代价函数  #相比于非正则化的逻辑回归代价函数，就是多了reg项  #其中reg项中的j是从1开始, 而不是0  def costReg(theta, X, y, learningRate):  theta = np.matrix(theta)  X = np.matrix(X)  y = np.matrix(y)  first = np.multiply(-y, np.log(sigmoid(X \* theta.T)))  second = np.multiply((1 - y), np.log(1 - sigmoid(X \* theta.T)))  #正规项中j从1开始  reg = (learningRate / (2 \* len(X))) \* np.sum(np.power(theta[:,1:theta.shape[1]], 2))  return np.sum(first - second) / len(X) + reg, gradientReg(theta, X, y) |

### 特征映射

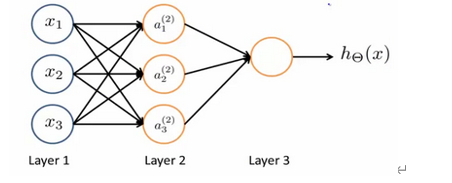
原案例中，只给了两个特征项，不好演示正则化，因此需要扩展特征项，使之包含一系列特征高次项。

|  |
| --- |
| degree = 5  x1 = data2['Test 1']  x2 = data2['Test 2']  data2.insert(3, 'Ones', 1)  for i in range(1, degree):  for j in range(0, i):  data2['F' + str(i) + str(j)] = np.multiply(np.power(x1, i-j) ,np.power(x2, j)) |

# 神经网络

## 原理

神经网络似乎是一个多层的逻辑回归，每一层增加一个1作为偏差单元（bias unit）,每层之间使用一组作为权重矩阵，



每一层的运算都是几组sigmoid函数运算，前后层的单元数量（也叫神经元数量）可以不同，考虑到前一层的最前面要补‘1’，偏差单元的维度应该是：后一层单元数量x（前一层单元数量+1）。

这个有点FPGA流水线的意思，每一层进行一个特定运算，向后传递，这个算法也确实被称为前向传播算法（FORWARD PROPAGATION）。

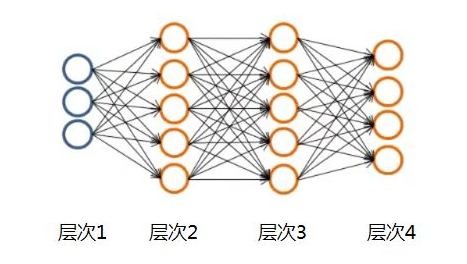
神经网络就是多层的逻辑回归，但是相比逻辑回归，比更先进。

## 代价函数

逻辑回归的代价函数：



神经网络就是对逻辑回归增加了维度的扩展：



用来标记层数，标记层的单元数，表示最终的分类数量（）。来标记某一层的行，来标记的列。

那么神经网络的代价函数为：

第一部分：

其实就是把各个分类的计算结果进行累加。也就是说，我们需要求得的参数，应该对每一个分类计算代价函数，并使得加总之后的结果最小。

第二部分：

表示各层的累加。

表示对某一层各行各列的累加。

表示行，第层的行数为下一层单元数量，即。从1开始因为不参与正则化，网上这么多的分析过程，你们就不舍得在这里加个括号吗。混淆视听！

表示列，从1开始是为了去除偏差项。

## 反向传播算法

### 理解

无论是前向传播，还是反向传播，都是需要计算梯度，然后通过高级优化算法计算最优。

但是对于神经网络，使用梯度下降算法以获取会导致计算过于庞大，不适用于工程。而应当使用反向传播算法作为神经网络的训练算法。

所以引入了一个神经元误差的概念，定义第层第个神经元的误差为：

其中:

那么可以由神经元误差推导代价函数偏导数：

已知：

那么：

### 神经元误差公式

以下图的4层神经算法为例：

图示

描述已自动生成

各层的误差值：

小tips（对Sigmoid函数求导）:

### 推导过程

推导过程就是围绕神经网络误差定义计算而来，对中间误差的计算会用到求导的链式法则：

表格, 信件

中度可信度描述已自动生成

**的推导过程**：

**的推导过程**（使用了链式法则，需要耐心）：

上面的转换运用了链式法则，其中有：

继续化简：

将结果向量化，考虑矩阵相乘的维度后，即得到的结果：