[Perceptron 2](#_Toc8180)

[1.1 perceptron model 2](#_Toc22090)

[1.2 perceptron learning strategy 3](#_Toc23858)

[1.3 perceptron algorithm 4](#_Toc9968)

[1.4 perceptron algorithm by python 5](#_Toc22986)

[1.4.1 algorithm overview 5](#_Toc20143)

[1.4.2 generate training data 6](#_Toc20112)

[1.4.3 Perceptron Class 7](#_Toc24171)

[1.4 perceptron algorithm convergence 8](#_Toc16084)

[1.4 perceptron dual form 8](#_Toc29278)

[Principal Component Analysis 9](#_Toc2916)

[2.1 basic theory 9](#_Toc20371)

[2.1.1 Mathematical Background 9](#_Toc5810)

[2.1.2 max variance theory 10](#_Toc515)

[2.1.3 practical example 11](#_Toc19187)

[2.2 algorithm step 11](#_Toc10405)

[2.2.1 overview 11](#_Toc6776)

[2.3 python code 15](#_Toc18164)

[2.2.1 overview 15](#_Toc5009)

[2.2.2 load\_data 15](#_Toc28602)

[2.2.3 overview 15](#_Toc11443)

[KNN(k-nearest neighbor) 15](#_Toc31519)

[2.1 basic theory 15](#_Toc21186)

[2.1.1 definition 15](#_Toc19027)

[2.1.1 k choose 16](#_Toc21159)

[2.1.1 distance metric 17](#_Toc27970)

[2.1.1 kNN vs K-Means 17](#_Toc16542)

[2.1.1 cited 18](#_Toc25397)

[2.1 code 18](#_Toc21211)

[2.1.1 data set preparation 18](#_Toc13658)

[2.1.1 sklearn knn prediction 19](#_Toc4730)

[2.1.1 hold out/cross-validation 19](#_Toc25207)

[2.1.1 knn from scratch 22](#_Toc7384)

[2.1 in practice 24](#_Toc3421)

[2.1.1 predict 24](#_Toc16580)

[2.1.1 insertion/smooth curve 24](#_Toc11225)

[2.1 improvement ways 25](#_Toc24109)

[Logistic Regression 26](#_Toc8747)

[2.1 basic theory 26](#_Toc25647)

[2.1.1 sample example 26](#_Toc22359)

[2.1.1 multi classify 31](#_Toc26806)

[2.1.1 al evaluation 32](#_Toc28866)

[2.1.1 Regularization 34](#_Toc28176)

[2.1 code 34](#_Toc6142)

[2.1.1 data 34](#_Toc19998)

[2.1.1 code 35](#_Toc946)

[2.1.1 sklearn example 37](#_Toc16392)

[2.1.1 cited 38](#_Toc1279)

# Perceptron

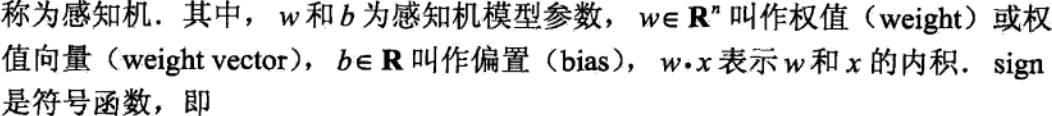
## 1.1 perceptron model

感知机是**二分类的线性分类模型**，其输入为实例的**特征向量**，输出为实例的类别取+1、-1。感知机将输入空间分为正负两类的**超平面**，称为**判别模型**。感知机的学习目的在于求出最佳超平面，由此导入基于误分类的**损失函数**。利用**随机梯度下降法**对损失函数进行最小化，求得感知机模型。

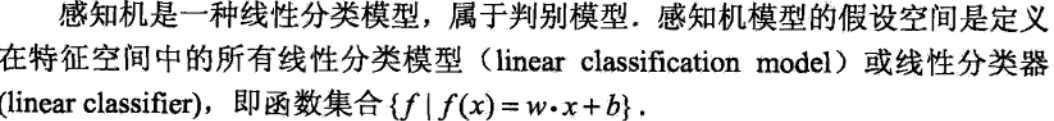
感知机是神经网络与支持向量机的基础。

定义 （**感知机**）： 假设输入空间(特征空间)是，输出空间是Y={1，-1}。输入表示实例的特征向量，对应于输入空间（特征空间）的点；输出表示实例的类别。由输入空间到输出空间的如下函数

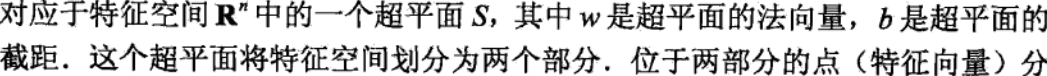
f(x)=sign(w.x+b)



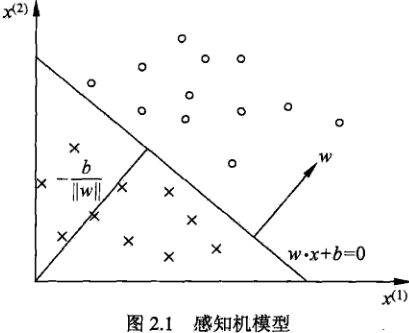




感知机的几何解释：线性方程w.x+b=0

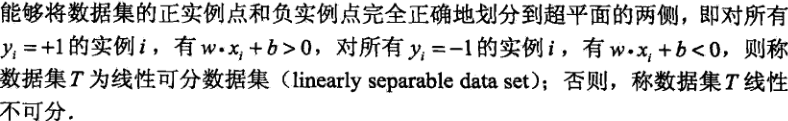




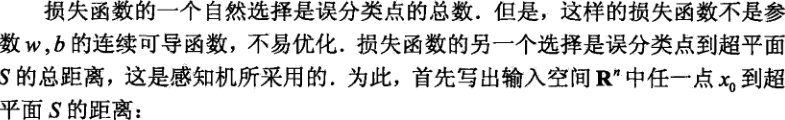


## 1.2 perceptron learning strategy

定义：**数据集的线性可分** 对于一个给定的数据集



假设数据是线性可分的，那如何找到这样一个超平面呢？即确定感知机模型参数w、b。

由此我们需要一个学习策略，即定义（**经验**）**损失函数**并将损失函数最小化。





误分类数据（本来y=1结果却识别为了-1，而-1识别为了1）





这样，假设超平面S的误分类点集合为M，那么所有的误分类点到超平面S的总距离为，不考虑，就得到感知机学习的损失函数.

给定数据集



损失函数定义为：

其中M为误分类点的集合，这个损失函数就是感知机学习的**经验风险函数。**

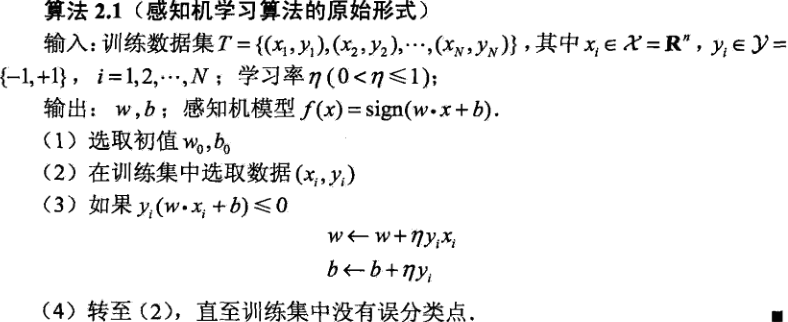
## 1.3 perceptron algorithm

感知机学习问题转化为了求损失函数的最优化问题。感知机学习算法是误分类驱动的，于是考虑使用，随机梯度下降法。

随机梯度下降法：

1. 随机选择一个超平面
2. 计算梯度 
3. 

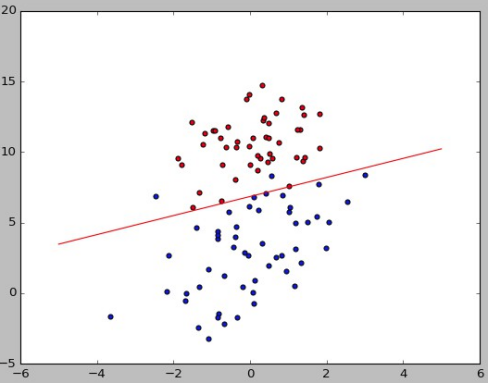
 



## 1.4 perceptron algorithm by python

### 1.4.1 algorithm overview

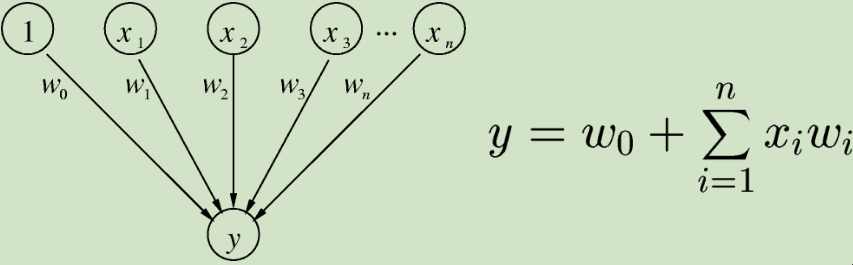
问题：使用感知机做二分类问题，数据集如下前两列为维数，最后一列为分类结果(为方便在准备数据的阶段，把0转化为了-1)。右图中红蓝点为数据的散点图，而红色的直线代表感知机生成的划分平面。

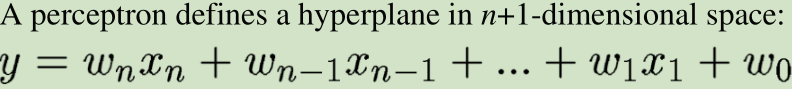
 

if \_\_name\_\_ == '\_\_main\_\_':  
 data\_path = '../dataset/perception/dataset.txt'  
 train\_data = generate\_data(data\_path)  
 epoch, eta, var\_num, batch\_size = 100, 0.1, 2, 20  
 p = Perception(var\_num)  
 p.sgd(train\_data, epoch, eta, batch\_size)  
 plot\_data\_scatter(train\_data, p.get\_weight(), p.get\_bias())

代码的总体思路：

1. 从dataset.txt读取数据，生成训练数据形式
2. 生成感知机类，以及随机梯度下降法参数，并训练网络
3. 画出训练数据集的散点图，以及最终生成的分类平面





展开形式如下：



例如, 二维空间(只有一个特征):

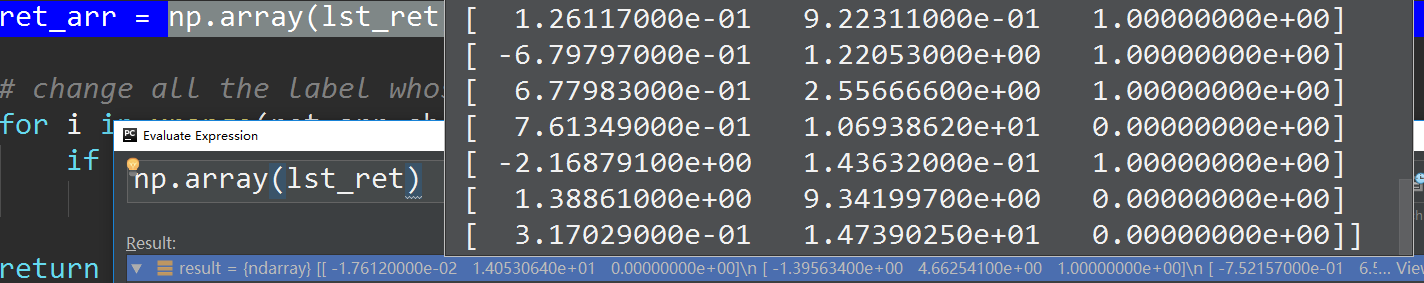
等价于直线分类，y=kx+b

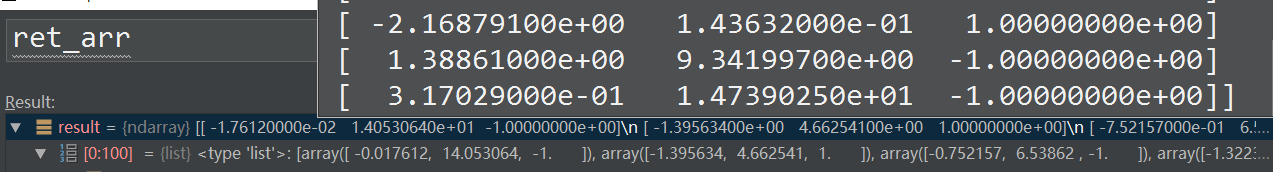
对于3D空间（两个特征，本文示例数据就是两个特征）：

等价于3D方程式：

### 1.4.2 generate training data

def generate\_data(data\_path):  
 lst\_data = get\_line\_lst(data\_path)  
  
 *# lst\_ret = []  
 # for item in lst\_data:  
 # lst\_ret.append([float(s) for s in item.split()])  
 # the following one line is equivalent to the above for loop* lst\_ret = [[float(s) for s in item.split()] for item in lst\_data]  
  
 ret\_arr = np.array(lst\_ret)  
 *# change all the label whose value is 0 to -1* for i in xrange(ret\_arr.shape[0]):  
 if ret\_arr[i][-1] == 0:  
 ret\_arr[i][-1] = -1  
  
 return ret\_arr





### 1.4.3 Perceptron Class

class Perception(object):  
  
 def \_\_init\_\_(self, var\_num):  
 *# self.w = np.random.randn(1, var\_num)* self.w = np.ones(var\_num)  
 self.b = 1  
 self.var\_num = var\_num  
 self.min\_error\_rate = 0.02  
  
 def train(self, train\_data, eta):  
 """  
 training model  
 **:param** train\_data: array like [[1, 2, 0], [1.1, 0.8, 1]]  
 **:param** eta: learning rate:  
 **:return** none:  
 """  
 for item in train\_data:  
 output = (np.dot(self.w, item[0:-1]) + self.b)\*item[-1]  
 if output <= 0:  
 self.w += eta \* item[-1] \* item[0:-1]  
 self.b += eta \* item[-1]  
  
 def sgd(self, train\_data, epoch, eta, batch\_size):  
 """  
 Training perception model by stochastic gradient descent  
 **:param** train\_data: 2D array like [[1.1, 2.3, -1]] the last  
 item -1 train\_date[0][-1] means label  
 **:param** epoch:  
 **:param** eta:learning rate  
 **:return**:none  
 """  
 for i in xrange(epoch):  
 np.random.shuffle(train\_data)  
 batch\_lst = [train\_data[k:k+batch\_size] for k in xrange(0, len(train\_data), batch\_size)]  
 for mini\_batch in batch\_lst:  
 self.train(mini\_batch, eta)  
  
 current\_error\_rate = self.get\_error\_rate(train\_data)  
 print 'epoch {0} current\_error\_rate: {1}'.format(i+1, current\_error\_rate)  
 print self.get\_current\_para()  
 if current\_error\_rate <= self.min\_error\_rate:  
 break  
  
 def get\_error\_rate(self, validate\_data):  
 all\_len = validate\_data.shape[0]  
 error\_len = 0  
 for item in validate\_data:  
 output = np.dot(self.w, item[0:-1]) + self.b  
 output = 1 if output >= 0 else -1  
 error = True if output != item[-1] else False  
 if error:  
 error\_len += 1  
  
 return float(error\_len) / all\_len  
  
 def get\_current\_para(self):  
 return self.w, self.b

## 1.4 perceptron algorithm convergence

算法的收敛性证明：先略过。

## 1.4 perceptron dual form

感知机的对偶形式。

对偶的目的：让计算变得简单高效

<https://www.jilp.org/cbp/Daniel-slides.PDF> (some illustration is very good)

# Principal Component Analysis

## 2.1 basic theory

PCA是一种无监督的学习方式，是一种很常用的降维方法。在数据信息损失最小的情况下，将数据的特征数量由n，通过映射到另一个空间的方式，变为k(k<n)。

### 2.1.1 Mathematical Background

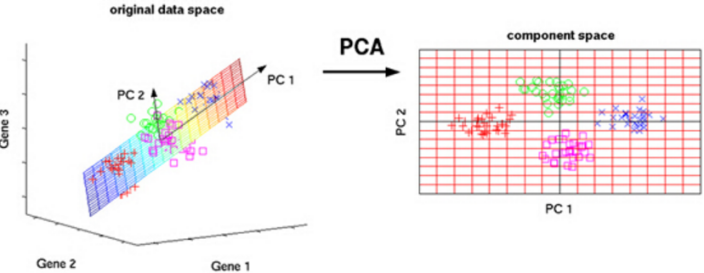
下表1是某些学生的语文、数学、物理、化学成绩统计：



首先，假设这些科目成绩不相关，也就是说某一科目考多少分与其他科目没有关系。那么一眼就能看出来，数学、物理、化学这三门课的成绩构成了这组数据的主成分（很显然，数学作为第一主成分，因为数学成绩拉的最开）。为什么一眼能看出来？因为坐标轴选对了！下面再看一组学生的数学、物理、化学、语文、历史、英语成绩统计，见表2，还能不能一眼看出来：



数据太多了，以至于看起来有些凌乱！也就是说，无法直接看出这组数据的主成分，因为在坐标系下这组数据分布的很散乱。究其原因，是因为无法拨开遮住肉眼的迷雾~如果把这些数据在相应的空间中表示出来，也许你就能换一个观察角度找出主成分。如下图1所示：



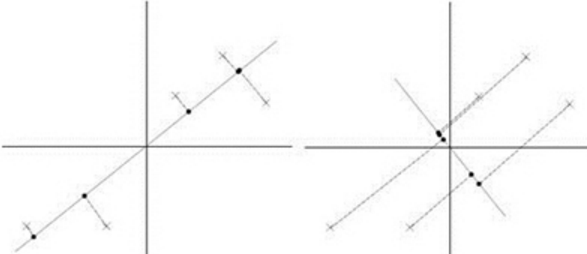
但是，对于更高维的数据，能想象其分布吗？就算能描述分布，如何精确地找到这些主成分的轴？如何衡量你提取的主成分到底占了整个数据的多少信息？所以，我们就要用到主成分分析的处理方法。

### 2.1.2 max variance theory

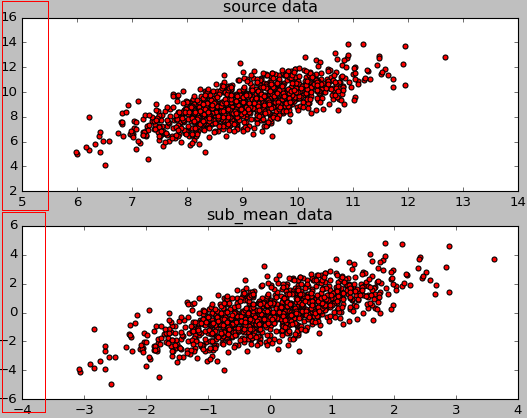
在信号处理中认为信号具有较大的方差，噪声有较小的方差，信噪比就是信号与噪声的方差比，越大越好。如前面的图，样本在u1上的投影方差较大，在u2上的投影方差较小，那么可认为u2上的投影是由噪声引起的。

因此我们认为，最好的k维特征是将n维样本点转换为k维后，每一维上的样本方差都很大。

比如我们将下图中的5个点投影到某一维上，这里用一条过原点的直线表示（数据已经中心化）：



假设我们选择两条不同的直线做投影，那么左右两条中哪个好呢？根据我们之前的方差最大化理论，左边的好，因为投影后的样本点之间方差最大（也可以说是投影的绝对值之和最大）

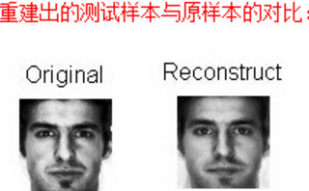


可以看到原始数据，以及求取平均值然后减去平均值，数据集相当于向下平移。

### 2.1.3 practical example

PCA将n个特征降维到k个，可以用来进行数据压缩，例如100维的向量最后可以用10维来表示，那么压缩率为90%。同样图像处理领域的KL变换使用PCA做图像压缩，人脸检测和匹配

...



可见测试样本为人脸的样本的重建误差显然小于非人脸的重建误差。

## 2.2 algorithm step

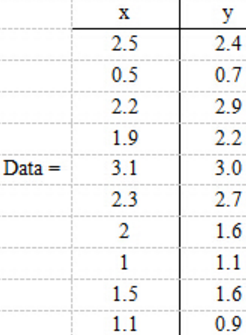
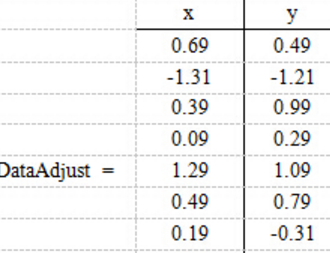
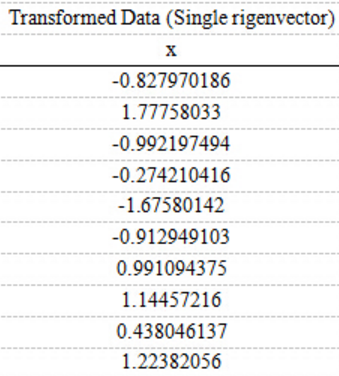
### 2.2.1 overview

The overview step :

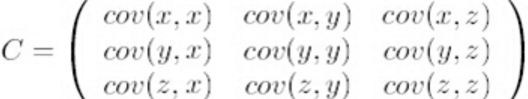
1. Remove the mean
2. Compute the covariance matrix
3. Find the eigenvalues and eigenvectors of the covariance matrix
4. Sort the eigenvalues from largest to smallest
5. Take the top N eigenvectors
6. Transform the data into the new space created by the top N eigenvectors

下面给个简单大概的示例：

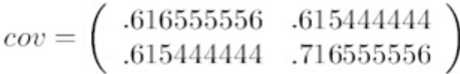
第一步，分别求x和y的平均值，然后对于所有的样例，都减去对应的均值。这里x的均值是1.81，y的均值是1.91，那么一个样例减去均值后即为（0.69,0.49），得到

第二步，求特征协方差矩阵，如果数据是3维，那么协方差矩阵是



这里只有x和y，求解得



对角线上分别是x和y的方差，非对角线上是协方差。协方差是衡量两个变量同时变化的变化程度。协方差大于0表示x和y若一个增，另一个也增；小于0表示一个增，一个减。如果ｘ和ｙ是统计独立的，那么二者之间的协方差就是０；但是协方差是０，并不能说明ｘ和ｙ是独立的。协方差绝对值越大，两者对彼此的影响越大，反之越小。协方差是没有单位的量，因此，如果同样的两个变量所采用的量纲发生变化，它们的协方差也会产生树枝上的变化。

第三步，求协方差的特征值和特征向量，得到



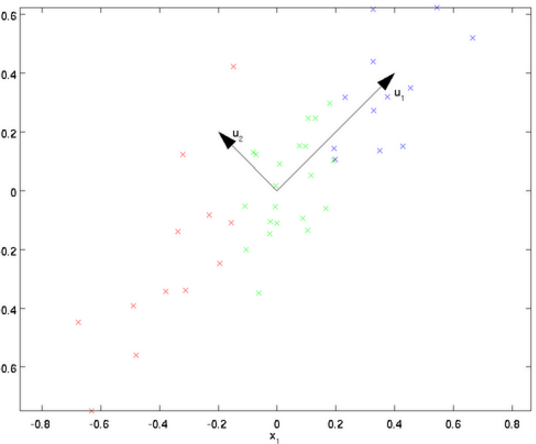
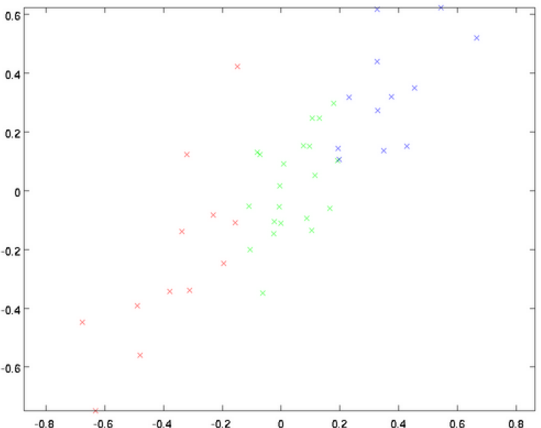
第四步，将特征值按照从大到小的顺序排序，选择其中最大的k个，然后将其对应的k个特征向量分别作为列向量组成特征向量矩阵。

这里特征值只有两个，我们选择其中最大的那个，这里是1.28402771，对应的特征向量是(-0.677873399, -0.735178656)T。

第五步，将样本点投影到选取的特征向量上。假设样例数为m，特征数为n，减去均值后的样本矩阵为DataAdjust(m\*n)，协方差矩阵是n\*n，选取的k个特征向量组成的矩阵为EigenVectors(n\*k)。那么投影后的数据FinalData为

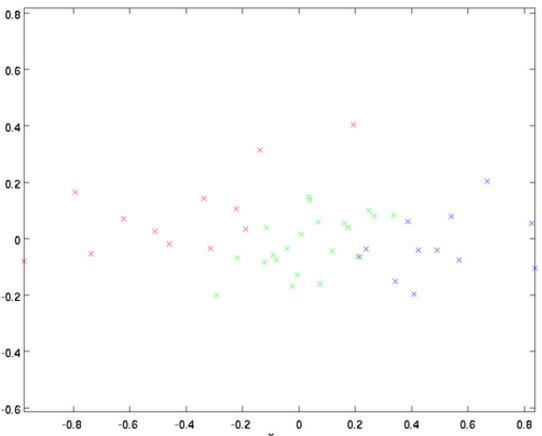
FinalData(10\*1) = DataAdjust(10\*2矩阵) \* 特征向量(-0.677873399, -0.735178656)T

得到的结果(见上面的图左右边的)

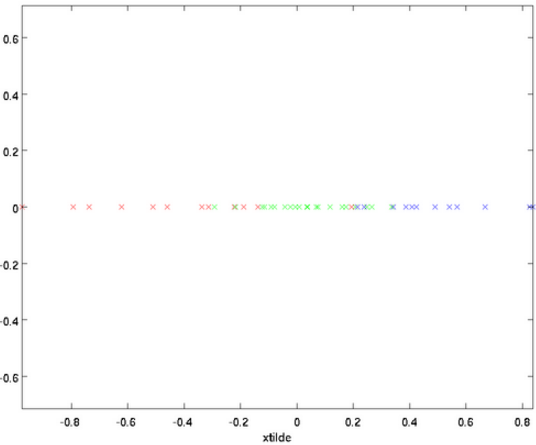


Rotating the Data

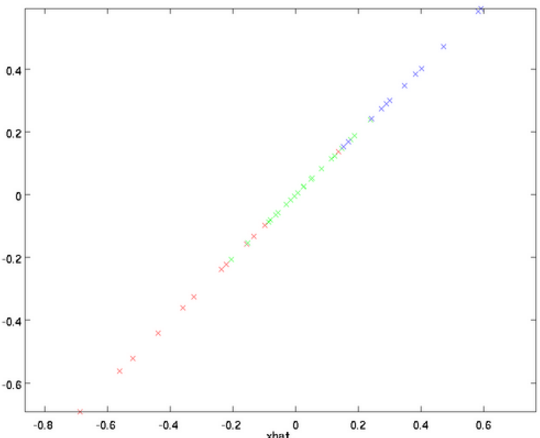
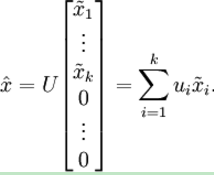
Thus, we can represent x in the  basis by computing

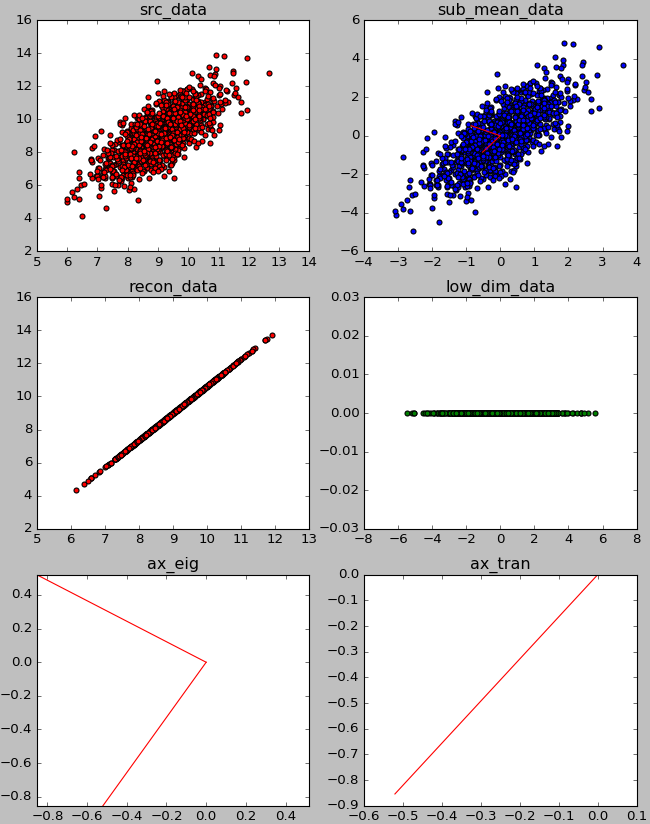


Reducing the Data Dimension



Recovering an Approximation of the Data





## 2.3 python code

### 2.2.1 overview

### 2.2.2 load\_data

### 2.2.3 overview

# KNN(k-nearest neighbor)

## 2.1 basic theory

### 2.1.1 definition

**K近邻算法**：给定一个训练数据集，对新的的输入实例，在训练数据集中找到与该实例最邻近的的K个实例，这K个实例的多数属于某个类，就把该实例分为这个类。

**K值选择**、**距离度量**、**以及分类决策**(一般多数表决)为K近邻算法的三个基本要素。

输入：训练数据集

KNN **falls in the supervised learning** family of algorithms. Informally, this means that we are given a labelled dataset consiting of training observations (x,y)and would like to capture the relationship between x and y. More formally, our goal is to learn a function h:X→Y so that given an unseen observation x, h(x) can confidently predict the corresponding output y.

The KNN classifier is also a **non parametric** and **instance-based** learning algorithm.

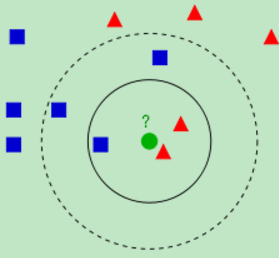
**Non-parametric** means it makes no explicit assumptions about the functional form of h, avoiding the dangers of mismodeling the underlying distribution of the data. For example, suppose our data is highly non-Gaussian but the learning model we choose assumes a Gaussian form. In that case, our algorithm would make extremely poor predictions.

**Instance-based** learning means that our algorithm doesn’t explicitly learn a model. Instead, it chooses to memorize the training instances which are subsequently used as “knowledge” for the prediction phase. Concretely, this means that only when a query to our database is made (i.e. when we ask it to predict a label given an input), will the algorithm use the training instances to spit out an answer.

**KNN is non-parametric, instance-based and used in a supervised learning setting.**

### 2.1.1 k choose

Wikipedia上的KNN词条中有一个比较经典的图如下：



从上图中我们可以看到，图中的有两个类型的样本数据，一类是蓝色的正方形，另一类是红色的三角形。而那个绿色的圆形是我们待分类的数据。

如果K=3，那么离绿色点最近的有2个红色三角形和1个蓝色的正方形，这3个点投票，于是绿色的这个待分类点属于红色的三角形。

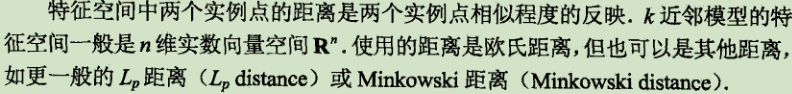
如果K=5，那么离绿色点最近的有2个红色三角形和3个蓝色的正方形，这5个点投票，于是绿色的这个待分类点属于蓝色的正方形。

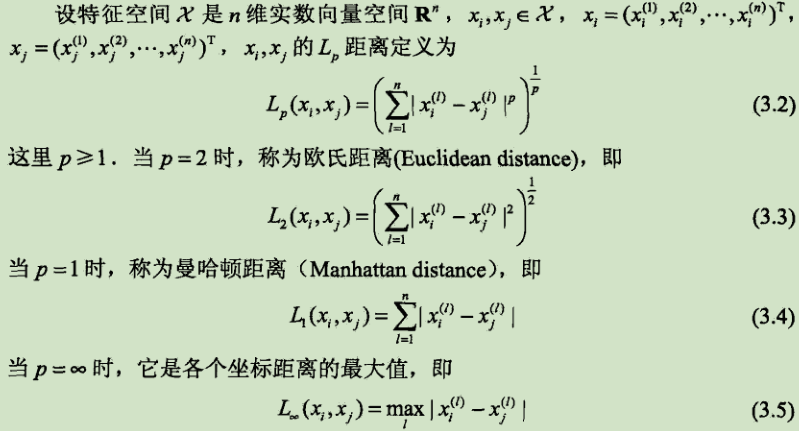
可见K值的选择对分类的结果还是有很大的影响。

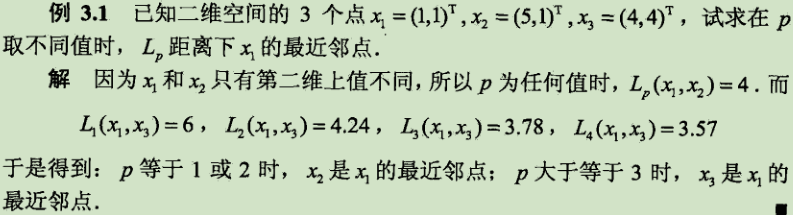
我们可以看到，机器学习的本质——是基于一种数据统计的方法！

**K应该是一个奇数，这样可以保证不会有平票**

### 2.1.1 distance metric







### 2.1.1 kNN vs K-Means

KNN算法和K-Means算法不同的是，K-Means算法用来聚类，用来判断哪些东西是一个比较相近的类型，而KNN算法是用来做归类的，也就是说，有一个样本空间里的样本分成很几个类型，然后，给定一个待分类的数据，通过计算接近自己最近的K个样本来判断这个待分类数据属于哪个分类。你可以简单的理解为由那离自己最近的K个点来投票决定待分类数据归为哪一类。

### 2.1.1 cited

https://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/

<http://coolshell.cn/articles/8052.html>

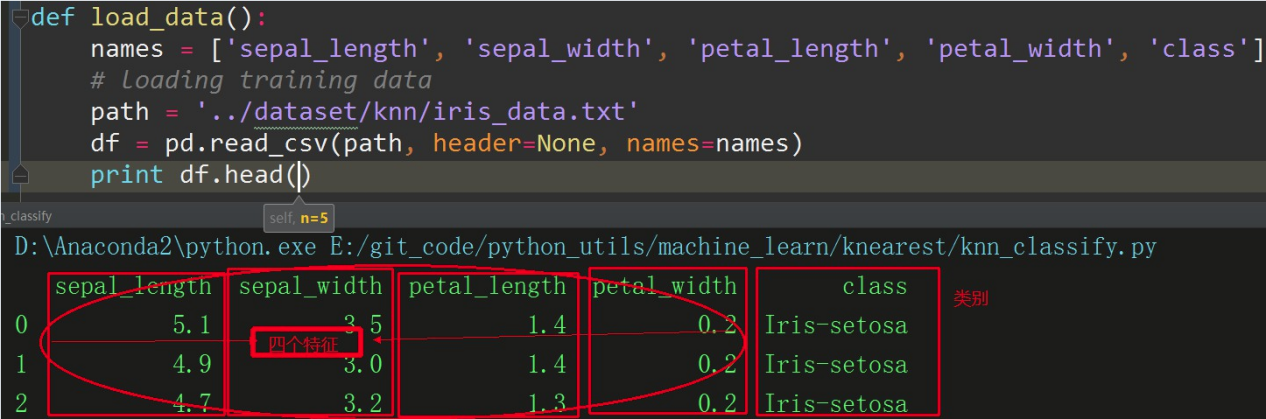
<https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/>

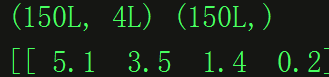
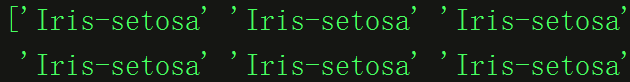
## 2.1 code

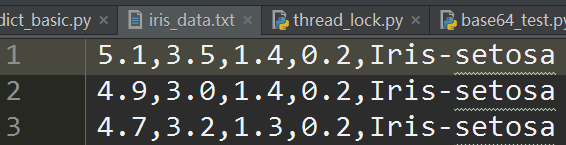
### 2.1.1 data set preparation

def load\_data():  
 names = ['sepal\_length', 'sepal\_width', 'petal\_length', 'petal\_width', 'class']  
 *# loading training data* path = '../dataset/knn/iris\_data.txt'  
 df = pd.read\_csv(path, header=None, names=names)  
 print df.head()

x = np.array(df.ix[:, 0: 4])  
 y = np.array(df['class'])print x.shape, y.shape





### 2.1.1 sklearn knn prediction

def predict():  
 x\_train, x\_test, y\_train, y\_test = load\_data()  
 k = 3  
 knn = KNeighborsClassifier(n\_neighbors=k)  
 knn.fit(x\_train, y\_train)  
 pred = knn.predict(x\_test)  
 print accuracy\_score(y\_test, pred)

### 2.1.1 hold out/cross-validation

**测试集选取**

测试集的选取：通常假设测试样本也是从样本真是分布中独立同分布采样得到，但是需注意，测试集应该尽可能与训练集互斥，即测试样本尽量不在训练集中出现、未在训练过程中使用（原因：希望得到泛化性能强的模型，就是学生做题时的举一反三能力）

已知：一个包含m个样例的数据集D={(x1,y1),(x2,y2),…,(xm,ym)}，既要训练又要测试。对D进行适当的处理，从中产生训练集S和测试集T。

**留出法(hold-out)**

方法：直接将数据集D划分为两个互斥的集合，其中一个集合作为训练集S，另一个作为测试集T，即.在S上训练出模型后，用T来评估其作为测试误差，作为对泛化误差的估计。

举个栗子：假定D包含1000个样本，将其划分为S包含700个样本，T包含300个样本，用S进行训练，如果模型在T上有90个样本分类错误，那么其错误率为，精度为1-30%=70%。

注意事项：

（1）训练/测试集的划分要尽可能保持数据分布的一致性，避免因数据划分过程引入额外的偏差而对最终结果产生影响。例如在分类任务中至少要保持样本的类别比例相似，尝采用分层采样的方法，即采样过程中保留类别比例。

（2）即便在给定训练/测试集的样本比例后，仍存在多种划分方式对初始数据集D进行分割。

因此，单次使用留出法得到的估计结果往往不够稳定可靠，在使用留出法时，一般要采用若干次随机划分、重复进行试验评估或取平均值作为留出法的评估结果。

在这里，还有一个窘境：若训练集S包含绝大多数样本，则训练处的模型可能更接近于用D训练出的模型，但由于T比较小，评估结果可能不够稳定准确，测试集小，评估结果的方差较大；若令测试集T多包含一些样本，则训练集S与差别更大了，被评估的模型与用D训练出的模型相比可能有较大的差别，从而降低了评估结果的保真性（fidelity）。

常见解决方法：将大约2/3~4/5的样本用于训练，剩余样本用于测试。

**交叉验证法（crossvalidation）/k折交叉验证（k-fold cross validation）**

方法：将数据集D划分为k个大小相似的互斥子集，即.每个子集Di都尽可能保持数据分布的一致性，即从D中通过分层采样得到。然后每次都用k-1个子集的并集作为训练集，余下的那个子集作为测试集；这样就可获得k组训练/测试集，从而可进行k次训练和测试，最终返回的是这k个测试结果的均值。

tips:

(1) 评估结果的稳定性和保真性很大程度上取决于k的取值，最常用取值为10，此时成为10折交叉验证；其他常用的k值有5、20.

(2) 为了减小因样本划分不同而引入的差别，k折交叉验证通常要随机使用不同的划分重复p次，最终的评估结果是这p次k折交叉验证结果的均值，例如常见的10次10折交叉验证。

(3) 特例：留一法：k=m，留一法使用的训练集与初始数据集相比只是少了一个样本，留一法中被实际评估的模型与期望评估的用D训练出的模型很相似，因此结果往往被认为比较准确。缺陷：数据集比较大事训练m个模型的计算开销可能过大。

2.3自助法（bootstrapping）

方法：以bootstrap sampling为基础，给定包含m个样本的数据集，我们对它进行采样产生数据集D’：每次随机从D中挑选一个样本，将其拷贝放入D’，然后将该样本放回初始数据集D中，使得该样本在下次采样时仍有可能被采到；这个过程重复m次以后，我们就得到了包含m个样本的数据集D’。将D’用作训练集，D\D’用作测试集；这样实际评估的模型与期望评估的模型都使用m个训练样本，而仍有数据总量约1/3的、没在训练集中出现的样本用于测试，这样的测试结果，也成为“包外估计”（out-of bag estimate）.

在这个过程中，样本在m次采样中始终不被采到的概率是(1-1/m)m，取极限得到

即通过自助采样，初始数据集D中约有36.8%的样本未出现在采样数据集D’中。

使用条件：

（1）在数据集较小，难以有效划分训练/测试集时很有用；

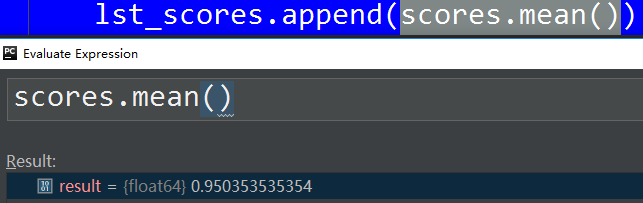
（2）可从初始数据集中产生多个不同的训练集，对集成学习等方法很有用。

总结：自助法产生的数据集改变了初始数据集的分布，这会引入估计偏差。在初始数据量足够时，留出法和交叉验证法更常用一些。



def cross\_validation():  
 x\_train, x\_test, y\_train, y\_test = load\_data()  
 k\_lst = list(range(1, 30))  
 lst\_scores = []  
  
 for k in k\_lst:  
 knn = KNeighborsClassifier(n\_neighbors=k)  
 scores = cross\_val\_score(knn, x\_train, y\_train, cv=10, scoring='accuracy')  
 lst\_scores.append(scores.mean())  
  
 *# changing to misclassification error* MSE = [1 - x for x in lst\_scores]  
 optimal\_k = k\_lst[MSE.index(min(MSE))]  
 print "The optimal number of neighbors is %d" % optimal\_k  
 *# plot misclassification error vs k  
 # plt.plot(k\_lst, MSE)  
 # plt.ylabel('Misclassification Error')* plt.plot(k\_lst, lst\_scores)  
 plt.xlabel('Number of Neighbors K')  
 plt.ylabel('correct classification rate')  
 plt.show()







### 2.1.1 knn from scratch

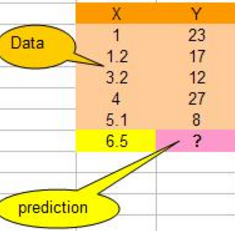
from collections import Counter  
import numpy as np  
class KnnScratch(object):  
  
 def fit(self, x\_train, y\_train):  
 self.x\_train = x\_train  
 self.y\_train = y\_train  
  
 def predict\_once(self, x\_test, k):  
 lst\_distance = []  
 lst\_predict = []  
  
 for i in xrange(len(self.x\_train)):  
 *# euclidean distance* distance = np.linalg.norm(x\_test - self.x\_train[i, :])lst\_distance.append([distance, i])  
  
 lst\_distance = sorted(lst\_distance)  
  
 for i in xrange(k):  
 idx = lst\_distance[i][1]  
 lst\_predict.append(self.y\_train[idx])  
  
 return Counter(lst\_predict).most\_common(1)[0][0]  
  
 def predict(self, x\_test, k):  
 lst\_predict = []  
 for i in xrange(len(x\_test)):  
 lst\_predict.append(self.predict\_once(x\_test[i, :], k))  
  
 return lst\_predict  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 x\_train = np.array([[1, 1, 1], [2, 2, 2], [10, 10, 10], [13, 13, 13]])  
 y\_train = ['aa', 'aa', 'bb', 'bb']  
 x\_test = np.array([[3, 2, 4], [9, 13, 11]])  
  
 knn = KnnScratch()  
 knn.fit(x\_train, y\_train)  
  
 print knn.predict\_once(x\_test[0], 2)  
 *# aa* print knn.predict(x\_test, 2)  
 *# ['aa', 'bb']*

## 2.1 in practice

### 2.1.1 predict

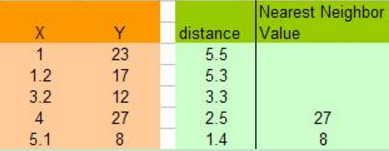
预测

假设我们有下面一组数据，假设X是流逝的秒数，Y值是随时间变换的一个数值（你可以想像是股票值）



那么，当时间是6.5秒的时候，Y值会是多少呢？我们可以用KNN算法来预测之。

这里，让我们假设K=2，于是我们可以计算所有X点到6.5的距离，如：X=5.1，距离是 | 6.5 – 5.1 | = 1.4， X = 1.2 那么距离是 | 6.5 – 1.2 | = 5.3 。于是我们得到下面的表：



注意，上图中因为K=2，所以得到X=4 和 X =5.1的点最近，得到的Y的值分别为27和8，在这种情况下，我们可以简单的使用平均值来计算：

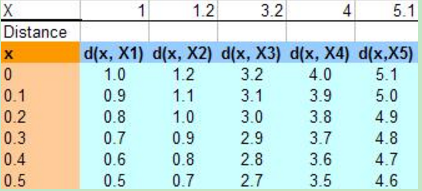
()/2=17.5, 于是，最终预测的数值为：17.5

### 2.1.1 insertion/smooth curve

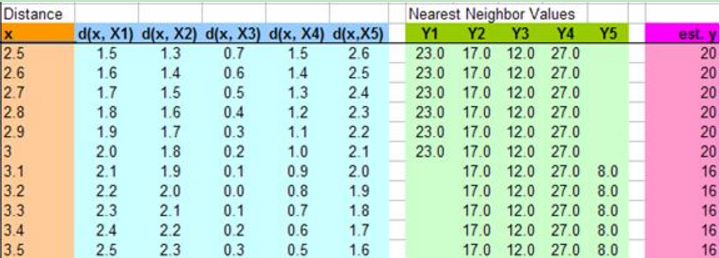
KNN算法还可以用来做平滑曲线用，这个用法比较另类。假如我们的样本数据如下（和上面的一样）：



要平滑这些点，我们需要在其中插入一些值，比如我们用步长为0.1开始插值，从0到6开始，计算到所有X点的距离（绝对值），下图给出了从0到0.5 的数据：



下图给出了从2.5到3.5插入的11个值，然后计算他们到各个X的距离，假值K=4，那么我们就用最近4个X的Y值，然后求平均值，得到下面的表：



于是可以从0.0, 0.1, 0.2, 0.3 …. 1.1, 1.2, 1.3…..3.1, 3.2…..5.8, 5.9, 6.0 一个大表，跟据K的取值不同，得到下面的图：

## 2.1 improvement ways

A simple and effective way to remedy skewed class distributions is by implementing weighed voting. The class of each of the K neighbors is multiplied by a weight proportional to the inverse of the distance from that point to the given test point. This ensures that nearer neighbors contribute more to the final vote than the more distant ones.

Changing the distance metric for different applications may help improve the accuracy of the algorithm. (i.e. Hamming distance for text classification)

Rescaling your data makes the distance metric more meaningful. For instance, given 2 features height and weight, an observation such as x=[180,70]x=[180,70] will clearly skew the distance metric in favor of height. One way of fixing this is by column-wise subtracting the mean and dividing by the standard deviation. Scikit-learn’s normalize() method can come in handy.

Dimensionality reduction techniques like PCA should be executed prior to appplying KNN and help make the distance metric more meaningful.

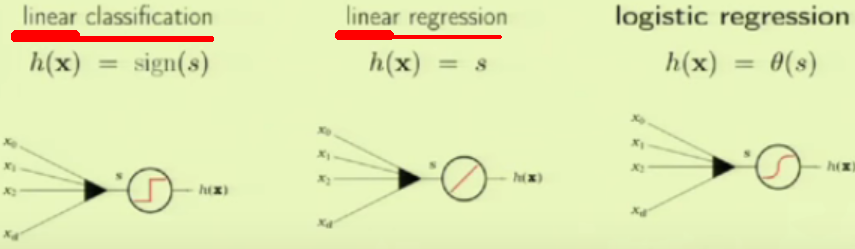
Approximate Nearest Neighbor techniques such as using k-d trees to store the training observations can be leveraged to decrease testing time. Note however that these methods tend to perform poorly in high dimensions (20+). Try using locality sensitive hashing (LHS) for higher dimensions.

# Logistic Regression

## 2.1 basic theory

### 2.1.1 sample example

Linear Regression （线性回归）考虑的是连续值（[0,1]之间的数）的问题，而Logistic Regression（逻辑回归）考虑的是离散值（例如只能取0或1而不能取0到1之间的数）的问题。举个例子，你需要根据以往季度的电力数据，预测下一季度的电力数据，这个时候需要使用的是线性回归，因为这个值是连续的，而不是离散的。而当你需要判断这个人抽烟还是不抽烟的问题时，就需要使用逻辑回归了，因为答案必然是抽烟或不抽烟这其中的一个，也就是离散值。



其中，线性分类(常见的感知机)

决策函数

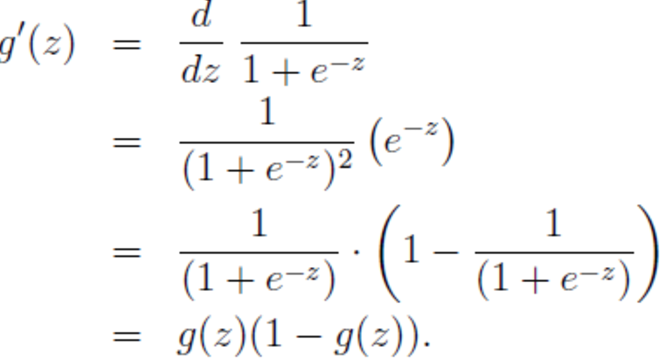
一个机器学习的模型，实际上是把决策函数限定在某一组条件下，这组限定条件就决定了模型的假设空间。当然，我们还希望这组限定条件简单而合理。而逻辑回归模型所做的假设是：



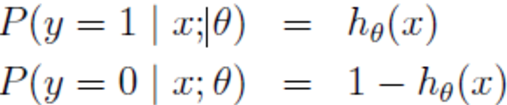


这里的 g(h)是上边提到的 sigmoid 函数，相应的决策函数为：

y=1,if P(y=1|x)>0.5(实际应用时特定的情况可以选择不同阈值，如果对正例的判别准确性要求高，可以选择阈值大一些，对正例的召回要求高，则可以选择阈值小一些)



那么，给定一个逻辑回归模型，如何来调整参数θ？首先我们假设：



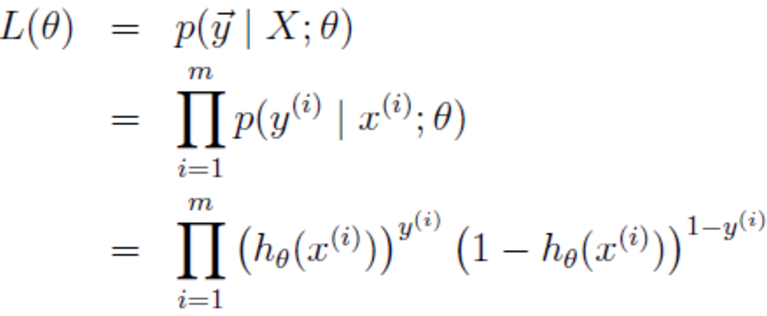
假设我们有n个独立的训练样本{(x1, y1) ,(x2, y2),…, (xn, yn)}，y={0, 1}。那每一个观察到的样本(xi, yi)出现的概率是：



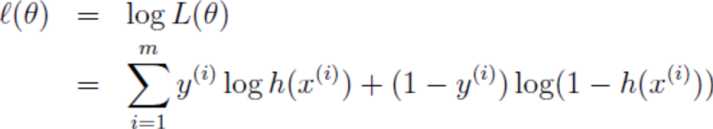
当y=1的时候，后面那一项是不是没有了，那就只剩下x属于1类的概率，当y=0的时候，第一项是不是没有了，那就只剩下后面那个x属于0的概率（1减去x属于1的概率）

为什么要 **似然函数**？(见概率论笔记)

因为每个样本都是独立的，所以n个样本出现的概率就是他们各自出现的概率相乘，假设生成m个训练样本相互独立，我们可以写出关于参数θ的似然函数：



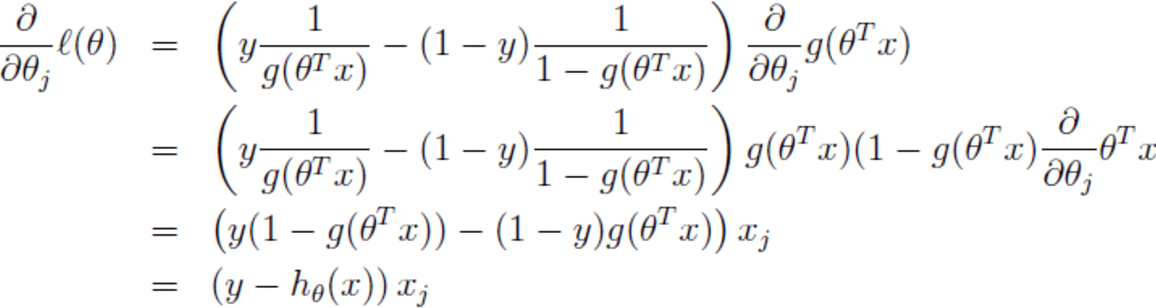
将它转换为log似然函数：



接下来我们如何最大化似然函数呢？跟之前的线性回归推导过程一样，我们可以使用梯度下降。即：

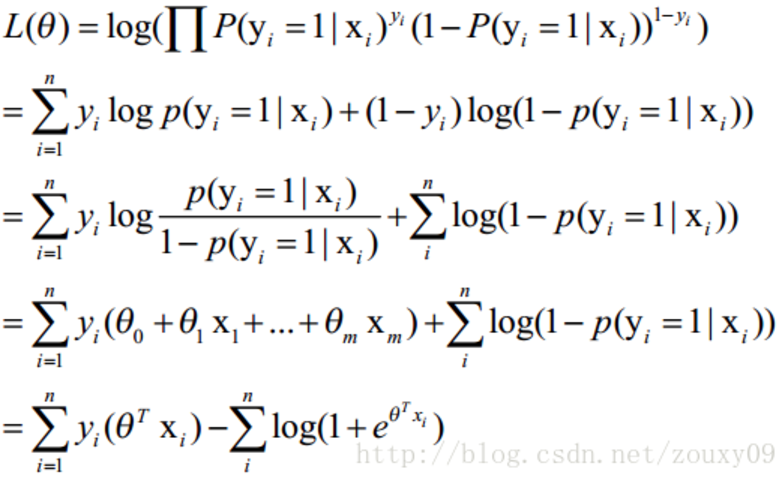


我们首先使用一个训练样本(x, y)，通过对似然函数求导，推导出随机梯度下降法则：

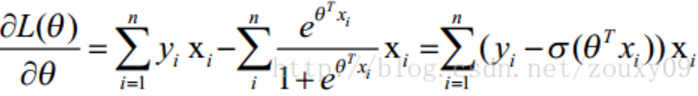


上式中，我们使用了g'(z)=g(z)(1-g(z))。最终得出随机梯度下降法则：



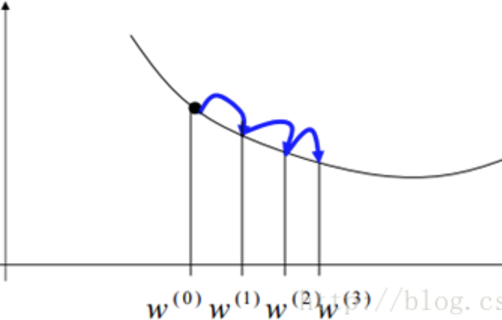


这时候，用L(θ)对θ求导，得到：

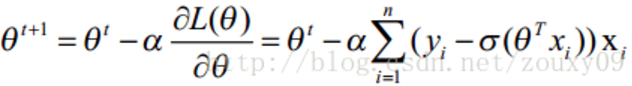


然后我们令该导数为0，你会很失望的发现，它无法解析求解。不信你就去尝试一下。所以没办法了，只能借助高大上的迭代来搞定了。这里选用了经典的梯度下降算法

Gradient descent 又叫 steepest descent，是利用一阶的梯度信息找到函数局部最优解的一种方法，也是机器学习里面最简单最常用的一种优化方法。它的思想很简单，和我开篇说的那样，要找最小值，我只需要每一步都往下走（也就是每一步都可以让代价函数小一点），然后不断的走，那肯定能走到最小值的地方，例如下图所示：



但，我同时也需要更快的到达最小值啊，怎么办呢？我们需要每一步都找下坡最快的地方，也就是每一步我走某个方向，都比走其他方法，要离最小值更近。而这个下坡最快的方向，就是梯度的负方向了



其中，参数α叫学习率，就是每一步走多远，这个参数蛮关键的。如果**设置的太多，那么很容易就在最优值附加徘徊，因为你步伐太大了**。例如要从广州到上海，但是你的一步的距离就是广州到北京那么远，没有半步的说法，自己能迈那么大步，是幸运呢？还是不幸呢？事物总有两面性嘛，它带来的好处是能很快的从远离最优值的地方回到最优值附近，只是在最优值附近的时候，它有心无力了。但如果**设置的太小，那收敛速度就太慢了**，向蜗牛一样，虽然会落在最优的点，但是这速度如果是猴年马月，我们也没这耐心啊。所以有的改进就是在这个学习率这个地方下刀子的。我**开始迭代是，学习率大**，**慢慢的接近最优值的时候，我的学习率变小就可以了**。所谓采两者之精华啊！

**随机梯度下降SGD (stochastic gradient descent)**

梯度下降算法在每次更新回归系数的时候都需要遍历整个数据集（计算整个数据集的回归误差），该方法对小数据集尚可。但当遇到有数十亿样本和成千上万的特征时，就有点力不从心了，它的计算复杂度太高。改进的方法是一次仅用一个样本点（的回归误差）来更新回归系数。

随机梯度下降算法的伪代码如下：

初始化回归系数为1

重复下面步骤直到收敛{

对数据集中每个样本

计算该样本的梯度

使用alpha xgradient来更新回归系数

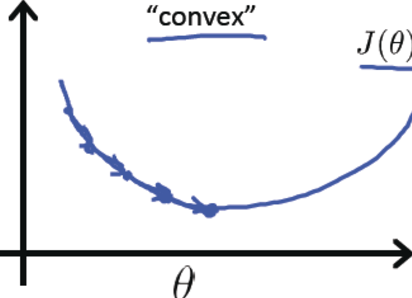
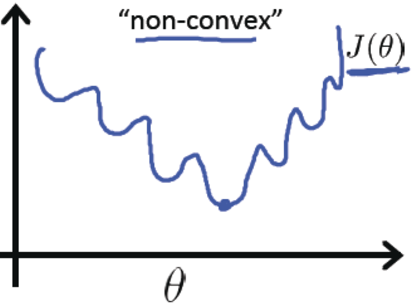
}

返回回归系数值

如果代价函数是凸函数，那么就存在全局最优解，方圆五百里就只有一个山峰，那命中注定了，它就是你要找的唯一了。但如果是非凸的，那么就会有很多局部最优的解，有一望无际的山峰，人的视野是伟大的也是渺小的，你不知道哪个山峰才是最高的.

也就是逻辑回归的Cost Function是“非凸”的，如下图所示：

对于凸函数来说局部最小值点即为全局最小值点，因此只要能求得这类函数的一个最小值点，该点一定为全局最小值点。



假设我们的样本是{x, y}，y是0或者1，表示正类或者负类，x是我们的m维的样本特征向量。那么这个样本x属于正类，也就是y=1的“概率”可以通过下面的逻辑函数来表示：

sigmoid函数是一个s形的曲线，它的取值在[0, 1]之间，在远离0的地方函数的值会很快接近0/1

总结：

1. 逻辑回归，模型已知参数未知，对给定样本出现概率最大值为条件求，模型最佳参数。

2. 模型已知参数未知，求解参数正好就是，最大似然求解

3. 随机梯度下降法，更新参数

### 2.1.1 multi classify

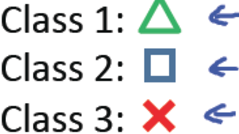
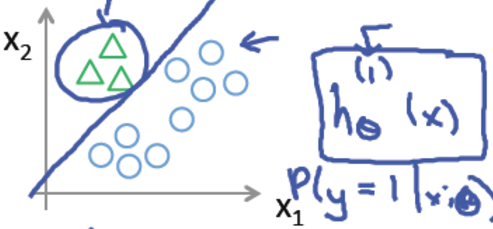
One-vs-all(one-vs-rest):

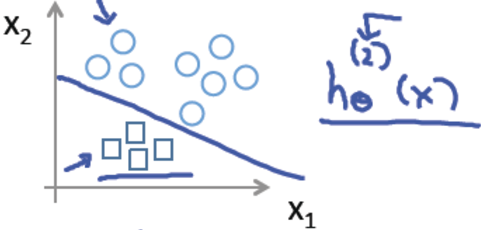
对于多类分类问题，可以将其看做成二类分类问题：保留其中的一类，剩下的作为另一类

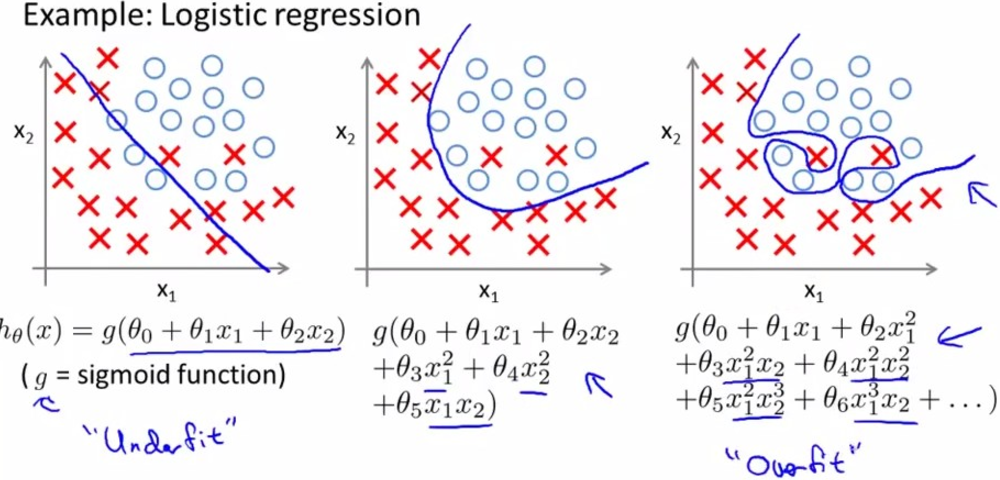
比如我想分成K类，那么就将其中一类作为positive，另（k-1）合起来作为negative，这样进行K个h(θ)的参数优化，每次得到的一个hθ(x)是指给定θ和x，它属于positive的类的概率。

例如：我将class1看成正例，那么class2与class3看成负例；相同的，将class2看成正例，class1与class3看成负例。。

可以分别计算其中一类相对于其他类的概率：

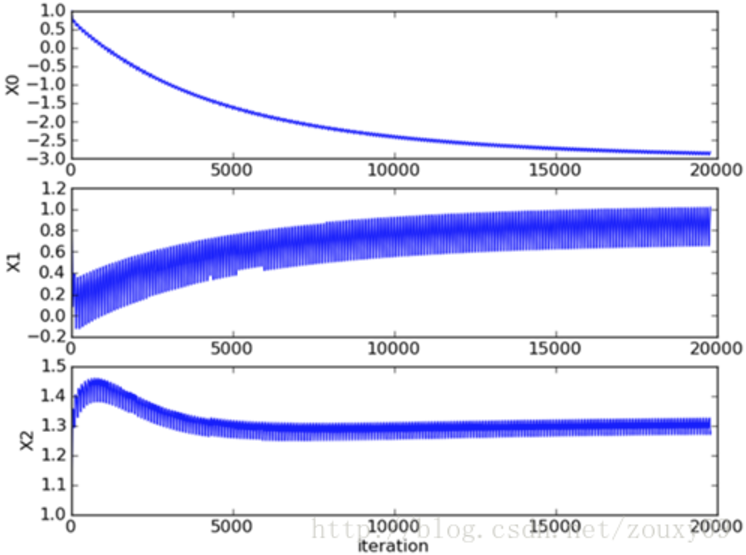
 





### 2.1.1 al evaluation

评价一个优化算法的优劣主要是看它是**否收敛**，也就是说参数是否达到稳定值，是否还会不断的变化？**收敛速度是否快**？

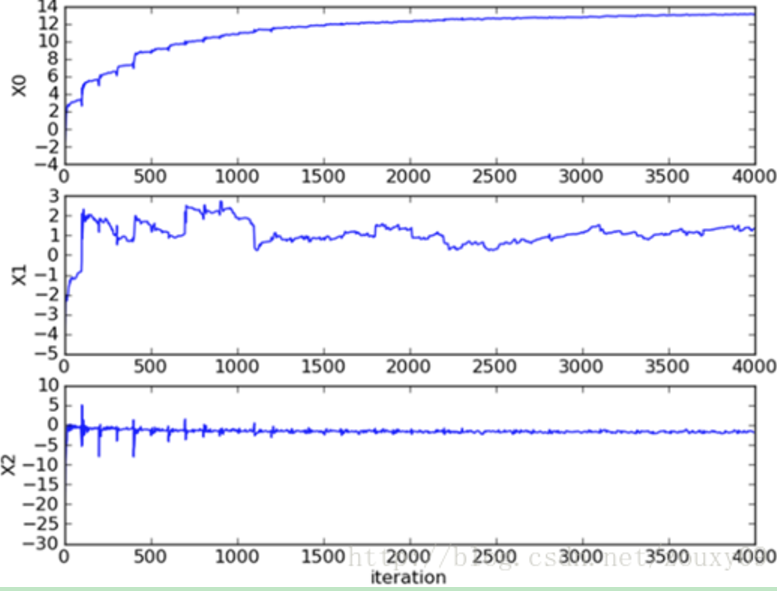


我们的数据库有100个二维样本，每个样本都对系数调整一次。上图展示了随机梯度下降算法在200次迭代所以共有200\*100=20000次调整）三个回归系数的变化过程。其中系数X2经过50次迭代就达到了稳定值。但系数X1和X0到100次迭代后稳定。而且可恨的是系数X1和X2还在很调皮的周期波动，迭代次数很大了，心还停不下来。产生这个现象的原因是存在一些无法正确分类的样本点，也就是我们的数据集并非线性可分，但我们的logistic regression是线性分类模型，对非线性可分情况无能为力。然而我们的优化程序并没能意识到这些不正常的样本点，还一视同仁的对待，调整系数去减少对这些样本的分类误差，从而导致了在每次迭代时引发系数的剧烈改变。对我们来说，我们期待算法能避免来回波动，从而快速稳定和收敛到某个值。

对随机梯度下降算法，我们做两处改进来避免上述的波动问题：

1）在每次迭代时，调整更新步长alpha的值。随着迭代的进行，alpha越来越小，这会缓解系数的高频波动（也就是每次迭代系数改变得太大，跳的跨度太大）。当然了，为了避免alpha随着迭代不断减小到接近于0（这时候，系数几乎没有调整，那么迭代也没有意义了），我们约束alpha一定大于一个稍微大点的常数项，具体见代码。

1. 每次迭代，改变样本的优化顺序。也就是随机选择样本来更新回归系数。这样做可以减少周期性的波动，因为样本顺序的改变，使得每次迭代不再形成周期性。



比较原始的随机梯度下降和改进后的梯度下降，可以看到两点不同：

1）系数不再出现周期性波动。2）系数可以很快的稳定下来，也就是快速收敛。这里只迭代了20次就收敛了。而上面的随机梯度下降需要迭代200次才能稳定。

### 2.1.1 Regularization

Regularization is a way of finding a good **bias-variance tradeoff** by tuning the complexity of the model. It is a very useful method to **handle collinearity** (high correlation among features), **filter out noise from data**, and eventually **prevent overfitting.**

## 2.1 code

### 2.1.1 data

def get\_data():  
 path = '../dataset/logistic\_regression/lr\_ml\_action.txt'  
  
 data = pd.read\_csv(path, delim\_whitespace=True,  
 names=['f1', 'f2', 'label'],  
 dtype={'A': np.float64, 'B': np.float64, 'C': np.int64})  
  
 *# add bias w0* data['f0'] = 1  
 print data.head()  
 features = ['f0', 'f1', 'f2']  
 return data[features].values, data.label.values

输出的前五行如下:

f1 f2 label f0

0 -0.017612 14.053064 0 1

1 -1.395634 4.662541 1 1

2 -0.752157 6.538620 0 1

3 -1.322371 7.152853 0 1

4 0.423363 11.054677 0 1

注意的是: 数据集添加逻辑回归的第一项即偏置W0。原始数据前五行如下:

其中前两列是特征，最后一列是label

-0.017612 14.053064 0  
-1.395634 4.662541 1  
-0.752157 6.538620 0  
-1.322371 7.152853 0  
0.423363 11.054677 0

### 2.1.1 code

class LogisticRegression(object):  
 def \_\_init\_\_(self):  
 self.\_map\_method()  
   
 def \_map\_method(self):  
 self.\_do\_train = {"gd": self.\_gd, "sgd": self.\_sgd}  
  
 def \_sigmoid(self, x):  
 return 1.0 / (1 + np.exp(-x))  
  
 def fit(self, X, Y, \*\*opt):  
 m, n = X.shape  
 self.\_w = np.ones((n, 1))  
 max\_iter = opt.get("max\_iter", 100)  
 alpha = opt.get("alpha", 0.01)  
 method = opt.get("method", "sgd")  
  
 for k in xrange(max\_iter):  
 try:  
 self.\_do\_train[method](X, Y, alpha)  
  
 print "iter %s error rate %s" % (k, self.\_get\_error\_rate(X, Y))  
 except KeyError:  
 raise ValueError('method error')  
  
 def \_sgd(self, X, Y, alpha):  
 """stochastic gradient descent"""  
 m, n = X.shape  
 for i in xrange(m):  
pred = self.\_sigmoid(np.dot(X[i, :], self.\_w))  
 error = Y[i] - pred  
 self.\_w= self.\_w + alpha \* np.matrix(X[i, :]).T \* error  
  
 def \_gd(self, X, Y, alpha):  
 """gradient descent"""  
 pred = self.\_sigmoid(X \* self.\_w)  
 error = Y - pred  
 self.\_weight = self.\_ + alpha \* X.T \* error  
  
 def \_get\_error\_rate(self, X, Y):  
 all\_num = len(Y)  
 error\_num = 0  
 for i in xrange(all\_num):  
 pred = self.\_sigmoid(np.dot(X[i, :], self.\_w)) > 0.5  
 if pred != bool(Y[i]):  
 error\_num += 1  
  
 return error\_num \* 1.0 / all\_num

注意：

1. 权重w的更新，分为梯度下降法和上面的随机梯度下降法，前者更新pred使用了整个数据集，于是考虑到使用随机梯度下降法。

def test\_lr():  
 X, Y = get\_data()  
  
 lr = LogisticRegression()  
 lr.fit(X, Y)

输出:

iter 0 error rate 0.28

iter 1 error rate 0.23

iter 2 error rate 0.22

iter 3 error rate 0.2

iter 4 error rate 0.19

iter 5 error rate 0.19

iter 6 error rate 0.18

iter 7 error rate 0.18

iter 8 error rate 0.16

iter 9 error rate 0.14

iter 10 error rate 0.14

iter 11 error rate 0.13

iter 12 error rate 0.12

### 2.1.1 sklearn example

### 2.1.1 cited

<http://blog.csdn.net/zouxy09/article/details/20319673>

<https://tech.meituan.com/intro_to_logistic_regression.html>

<http://blog.yhat.com/posts/logistic-regression-python-rodeo.html> (不错的示例)

<http://www.dummies.com/programming/big-data/data-science/using-logistic-regression-in-python-for-data-science/>

<http://dataaspirant.com/2017/05/15/implement-multinomial-logistic-regression-python/>

<https://www.codeproject.com/Articles/821347/MultiClass-Logistic-Classifier-in-Python>

<http://scikit-learn.org/stable/tutorial/statistical_inference/supervised_learning.html>

<http://scikit-learn.org/stable/tutorial/basic/tutorial.html>

<http://scikit-learn.org/stable/auto_examples/linear_model/plot_iris_logistic.html>

<http://www.jianshu.com/p/c235da8c1878>

<https://www.zhihu.com/question/20447622>