# AI学习笔记

## AI学习的基本概念

没有AI，没有人工智能，当前的计算机处理不存在智能，所谓的AI，所谓的人工智能，只不过是面对一些特定问题时，采取了与传统计算机处理问题不同的思路和方法。

**1：传统计算机对问题的解决**

通过定义流程，数据结构，操作命令，设置满足或不满足的条件来对特定的问题进行处理，例如，我们有一堆红色和白色的球混合在一起，我们要进行区分，传统上我们的做法是下面的样子：

检查这些球，人为归纳出红色的球具有的特征，归纳出白色的球具有的特征，然后让计算机根据这些特征进行判断，如果 一个球具有归纳出的红球特征，则认为这个球是红色，如果一个球具有白球特征，则认为这个球是白色。

当计算机的程序确定以后，只要是具有红球特征的球被传递计算机后，即使该球是白色的，计算机仍然会认为他是红色的。

**2：当前大热的人工智能是如何解决这类问题**

我们先找很多的白球和红球，并且做好标记，比如说A球是红球，B球是白球，然后将这些球传递给计算机，让计算机代替人进行这些球的特征的归类；当计算机利用特殊的算法进行大量的归类以后，我们传递一个新的球给计算机后，计算机就能判断这个球是红色或白色（当然这样仍然存在偏差）

所以这是一个数学或算法问题。不是传统意义或科幻电影中所讲的人工智能。

当然由于互联网的发展，产生了大量的数据，从这些数据中归纳总结出一定的规则，然后将这些规则应用到新的分析判断上，产生了非常好的效果。

所以当前所讲的AI或人工智能，都属于机器学习数据特征并总结归纳的一部分。

## 机器如何进行学习

计算机通过人工定义的算法进行学习。这样的算法有很多，例如KNN，决策树等。

从大的方面又划分位有监督学习算法和无监督学习算法。

有监督学习算法：对于数据，由人先给出数据对应的结果，然后由机器根据这些数据学习对应的特点和规律，学习完成后，可以应用到新的没有对应结果的数据上并可以对新数据进行处理。

无监督学习算法：对于数据，完全让机器根据数学规律进行统计分析，人不参与对数据标记。

## 总结：

机器学习就是：根据给定的数据特征和结论，利用数学的方法，让计算机找出一个这样的函数，这个函数的输入就是数据特征，输出就是对应结论。当找到这个函数以后，如果有新的数据，则将新的数据代入到这个函数中，从而计算出对应的结论。

# 机器学习算法

在研究机器学习算法之前，先明确一些基本的概念

1：数据特征：用于描述一条数据的一些特征属性，例如描述一个人，可以用身高，体重，姓名的属性来表述，这些数据就是数据的特征。

2：数据类别：根据数据的特征进行明确的分类。比如说身高1.7，体重60KG的这个人是小明。不满足上面这些特征的人不是小明。那么“是小明”和“不是小明”则被称作分类。

机器学习研究中一个很重要的问题就是解决分类问题，即根据数据的特征解决数据是属于哪个类别。机器学习中，有下面一些很经典的分类算法：

1. KNN紧邻算法
2. 决策树算法
3. 贝叶斯分类
4. Logistic回归

下面先从最简单的KNN紧邻算法开始说明

## KNN算法

KNN算法可以用来解决多种类别的分类问题，该算法描述如下：

已知数据集合X，X包括若干不同的样本{x1,x2,x3,x4…Xn}和对应的分类结果Y，Y包括{y1,y2,y3…yn}，当出现新的数据A时，求A数据对应的分类结果。

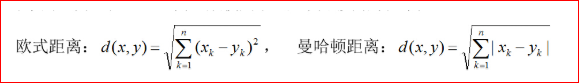
对于这样一个问题，KNN算法通过以下方法进行解决：

1：计算数据A与样本集合X中所有样本之间的距离，然后获取距离最小（特征最接近）的K个样本数据对应的分类标签{y1,y2,y3…}

2：从这分类标签中选择最多的分类标签为A数据的分类标签。

KNN算法在执行时一般K的选择不会超过20.

KNN算法计算数据A与样本集合X中不同的数据距离时，可以采取以下公式进行

即对应的特征差平方和开方得到。

KNN算法python代码实现（使用MINIST数据集测试验证）

**import** sys  
**import** os  
**import** struct  
**import** numpy **as** np  
**import** operator  
**import** cv2  
**from** matplotlib **import** pyplot **as** plt  
  
**def** loadMnistImage(path):  
 **with** open(path, **'rb'**) **as** imgfp:  
 magic, num, row, col = struct.unpack(**'>IIII'**, imgfp.read(16))  
 imglen = row \* col  
 imgs = np.fromfile(imgfp, dtype=np.uint8).reshape(num, imglen)  
 *#read image from file and show it  
 #npfirstImg = np.array(list(imgfp.read(imglen))).reshape(row, col)  
 #plt.imshow(npfirstImg)  
 #plt.show()* **return** imgs  
**def** loadMnistLable(path):  
 **with** open(path, **"rb"**) **as** labelfp:  
 magic, num= struct.unpack(**'>II'**, labelfp.read(8))  
 labels = np.fromfile(labelfp, dtype=np.uint8)  
 **return** labels  
  
**def** GetPredictLableByKNN(trainImages, trainLabels, predictImage, knum=20):  
 predictList = {}  
 trainImagesNum = len(trainImages)  
 **for** trainImagesIdx **in** range(trainImagesNum):  
 *#calc the distance for preicitImage and trainImage* difSum = 0.0  
 **for** pixIdx **in** range(len(predictImage)):  
 difSum += (np.int(predictImage[pixIdx]) - np.int(trainImages[trainImagesIdx][pixIdx])) \*\* 2  
 *#sqrt* distance = np.sqrt(difSum)  
 sortPredList = sorted(predictList.items(), key = operator.itemgetter(1), reverse=**True**) *#from big to little* **if** len(sortPredList) < knum:  
 predictList.update({trainImagesIdx:distance})  
 **continue  
 if** distance < sortPredList[0][1]:  
 rmvKey = sortPredList[0][0]  
 predictList.pop(rmvKey)  
 predictList.update({trainImagesIdx:distance})  
 *#check the labels and return* predictLables = {}  
 **for** item **in** predictList.items():  
 label = trainLabels[item[0]]  
 **if** label **in** predictLables.keys():  
 predictLables.update({label: predictLables[label]+1})  
 **else**:  
 predictLables.update({label:1})  
  
 lablesRsList = sorted(predictLables.items(), key=operator.itemgetter(1), reverse=**True**)  
 print(lablesRsList)  
 **return** lablesRsList[0][0]  
  
**def** main():  
 trainLables = loadMnistLable(**"E:\\mlDataSet\\mnist\\train-labels.idx1-ubyte"**)  
 trainImgs = loadMnistImage(**"E:\\mlDataSet\\mnist\\train-images.idx3-ubyte"**)  
  
 testLables = loadMnistLable(**"E:\\mlDataSet\\mnist\\t10k-labels.idx1-ubyte"**)  
 testImgs = loadMnistImage(**"E:\\mlDataSet\\mnist\\t10k-images.idx3-ubyte"**)  
  
 errPredict = 0  
 **for** testImgItem **in** range(len(testImgs)):  
 predictRs = GetPredictLableByKNN(trainImgs, trainLables, testImgs[testImgItem], 20)  
 **if** predictRs != testLables[testImgItem]:  
 errPredict = errPredict + 1  
  
 print(**"Err predict num="**, errPredict)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

说明：KNN算法有较好的预期结果，但需要一份比较有效的数据集合，同时KNN算法在预测时需要花费大量的时间计算待预测样本与目标数据集合之间的关系，对预测效率和内存都有比较大的占用。

## 机器学习之线性回归

什么是线性回归？

回归问题一般用于预测输入与输出变量之间的关系。线性回归是用线性函数预测输入与输出之间的关系。下面举个例子说明：

例如房屋面积和房屋价格之间的关系。

如果我们有某个地区房屋面积和房屋架构的大量数据，那么给出一个新的房屋后，该新房屋的价格是多少。

线性回归即可以用来解决这类问题。

即认为输出结果与输入变量之间满足线性函数关系

Y=wX+b。

其中在以上公式中，Y是输出结果，X是输入变量，w和b是参数。

当我们有大量的数据时，通过这些数据求出w和b的取值后，则能得到完整的Y=wX+b表示，这样当以后有新的输入数据时，将新的数据代入该公式，就可以得到对应结果。

以上是对线性回归的一个简单解释。

那么如何根据已有数据求出对应参数w和b。

机器学习的做法就是尝试给w和b设置不同的取值，然后根据训练数据中的输入数据X求出输出结果Y，然后用求出的这个结果与训练数据中真实的结果进行比较。如果这个差异是最小的，那么就认为w和b的取值已经找到。

预测结果和真实结果的比较在这里被定义为一个函数被称为损失函数（损失函数可以根据业务场景的不同，被定义为不同的形式），机器学习的过程就是求取损失函数最小。

在线性回归过程中，损失函数经常使用最小二乘法。实际值与预测值之差的平方和最小。

下面通过python代码实现线性回归。

数据集的获取和构造

在这里我们使用sklearn库中的波士顿房价数据集。

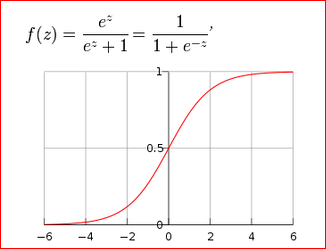
波士顿房价数据集合是一个包含506条记录的数据集合，每条记录会有13个特征，对应价格为一个1维的数据，因此我们的w参数是一个13行1列的矩阵。基于tensorflow的代码如下：

import sys  
import os  
import numpy as np  
from sklearn import datasets  
import tensorflow as tf  
  
def inference(X, W, b):  
 #if X is an matrix of 500 row and 13 rol, the W should is an 13 row and 1 col  
 #the result is an 500 row and 1 col  
 return tf.matmul(X, W) + b  
  
def loss(X, Y, W, b):  
 #while we execute inference, the result is and 500 row and 1 col, so we need change it to 1 row and 500 col because  
 #the labes Y is 1 row and 500 col, other we can not execute tf function because the shape is not the same  
 Y\_predicted = tf.transpose(inference(X, W, b))  
 return tf.sqrt(tf.reduce\_sum(tf.squared\_difference(Y, Y\_predicted)))  
  
def inputs():  
 boston = datasets.load\_boston()  
 return tf.to\_float(boston['data']), tf.to\_float(boston['target'])  
  
def train(total\_loss):  
 learning\_rate = 0.000001  
 return tf.train.GradientDescentOptimizer(learning\_rate).minimize(total\_loss)  
  
def main():  
 X, Y = inputs()  
 W = tf.Variable(tf.zeros([X.get\_shape()[1], 1]), name="weights")  
 b = tf.Variable(0., name='bias')  
 initator = tf.global\_variables\_initializer()  
 total\_loss = loss(X, Y, W, b)  
 train\_op = train(total\_loss)  
 train\_step = 1000  
 with tf.Session() as sess:  
 sess.run(initator)  
 for step in range(train\_step):  
 sess.run(train\_op)  
 print("Loss=", sess.run(total\_loss))  
  
 #if you want interfence new result, you can do it like this  
 rs = inference(X[0:1], W, b)  
 print("predict value =", sess.run(rs), ",real value=", sess.run(Y[0]))  
  
if \_\_name\_\_=='\_\_main\_\_':  
 main()

## 机器学习之逻辑回归

逻辑回归是一个概率模型，可以用来解决二元分类问题，在逻辑回归中，我们认为预测结果是通过sigmod函数运算得到。

Sigmod函数的形式如下：



在整个实数空间上，sigmod函数的运算结果落在0到1的区间范围之内，当应用到二元分类问题时，可以做下面这样的认为，即如果运算结果大于0.5，则认为yes，如果小于0.5则认为分类是no，当使用逻辑回归时，由于对应问题是一个概率问题，因此在损失函数的选择上，经常选择交叉熵作为损失函数。

交叉熵的函数形式如下：



交叉熵的更深入内容可以参考以下博客内容

https://blog.csdn.net/tsyccnh/article/details/79163834

下面使用sklearn数据集合中的乳腺癌数据集，基于tensorflow实现。

乳腺癌数据集是一个包含569条记录的小数据集合，该数据集合中每条记录有30个特征。

import os  
import sys  
import numpy as np  
import tensorflow as tf  
from sklearn import datasets  
  
def input():  
 breastData = datasets.load\_breast\_cancer()  
 lablelen = len(breastData['target'])  
 return tf.to\_float(breastData['data']), tf.reshape(tf.to\_float(breastData['target']), [lablelen, 1])  
  
def inference(X, W, b):  
 return tf.sigmoid(tf.matmul(X, W) + b)  
  
def loss(X, Y, W, b):  
 combine\_input = tf.matmul(X, W) + b  
 return tf.reduce\_mean(tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=combine\_input, labels=Y))  
  
def train(total\_loss):  
 learning\_rate = 0.000001  
 return tf.train.GradientDescentOptimizer(learning\_rate).minimize(total\_loss)  
  
def main():  
 X, Y = input()  
 W = tf.Variable(tf.zeros([X.get\_shape()[1], 1]), name="weights")  
 b = tf.Variable(0., name='bias')  
 print(W, b)  
 print(X, Y)  
  
 initator = tf.global\_variables\_initializer()  
 total\_loss = loss(X, Y, W, b)  
 train\_op = train(total\_loss)  
 train\_step = 1000  
 with tf.Session() as sess:  
 sess.run(initator)  
 for step in range(train\_step):  
 sess.run(train\_op)  
 print("loss=", sess.run(total\_loss))  
 #after finish train, you can prediect new data with following code  
 prers = inference(X[0:1], W, b)  
 real = Y[0:1]  
 print("preRs=", sess.run(prers), "real=", sess.run(real))  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

## 机器学习之决策树模型

在实现决策树模型之前，先需要了解信息的熵，熵被定义为信息的期望值，下面说明信息熵的计算方法：

假如事件发生的可能性包括A，B，C三类；那么计算分类A的熵按照以下方式

L(A) = -log2P(A)

其中L(A)为分类A的熵。P(A)为事件发生后是A类的概率（可能性）。那么所有类别的熵计算方式如下：

H = -∑P(xi)log2P(xi) 其中P(xi)为i类别的概率。

如果给出一份待训练的数据集合，则可以根据以上公式计算出该数据集的信息熵，计算规则如下：

1：根据分类标签，计算不同分类下的概率P

2：利用最大熵计算原则计算该数据集的熵。

以上是信息熵的基本概念。

下面说明信息增益的概念：

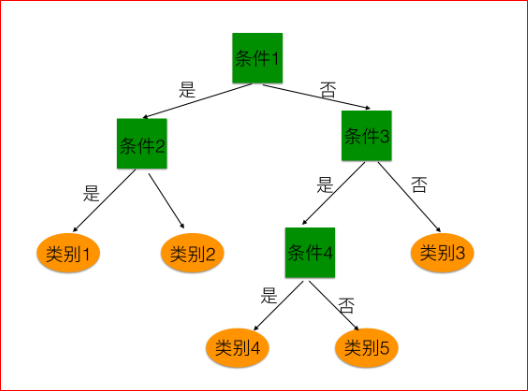
信息增益是对信息前后变化量的描述，其计算方式为

数据变化前的信息熵与数据变化后的信息熵之差，如果信息增益越大，说明变化后的数据信息熵减少，其数据更加有序，价值更高。

**什么是决策树。**

决策树是一个树形结构，该树的每个分支对应一个判断条件，从树的根节点出发，通过一层又一层的分支进行判断，最终行进到叶子节点，该叶子节点对应哪个决策结果。这样的一个过程被称为决策树（非常类似于程序设计中的流程图）

如下图：



**如何构造决策树**

构造决策树的过程如下

1：选取数据集中的某个特征，根据该特征将数据集划分为两部分（例如：我们的数据集中有特征A，A的取值位a1和a2，那么我们可以根据特征A将数据集合划分为A取值为a1的子集1和A取值为a2的子集2），然后我们计算划分为子集后数据的信息增益是多少。

2：对每个特征都做步骤1的这种处理。最后检查看根据那个特征划分子集后，数据集的信息增益最大。

3：根据信息增益最大的特征构造决策树的第一层节点。

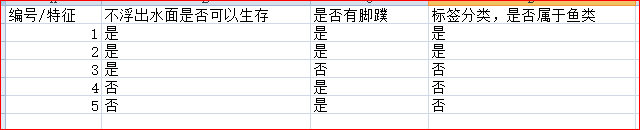
4：然后在子集上继续利用该原则，根据数据特征进行决策树第二层的构造。

5：如此递归，直到待判断的数据集属于同一个类别为止，这样一颗完整的决策树结构就构造成功。

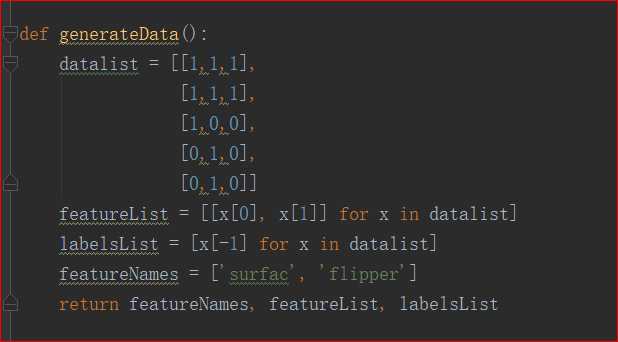
这个决策树后续就可以用在对新数据类别的预测上。

**决策树python例子：**

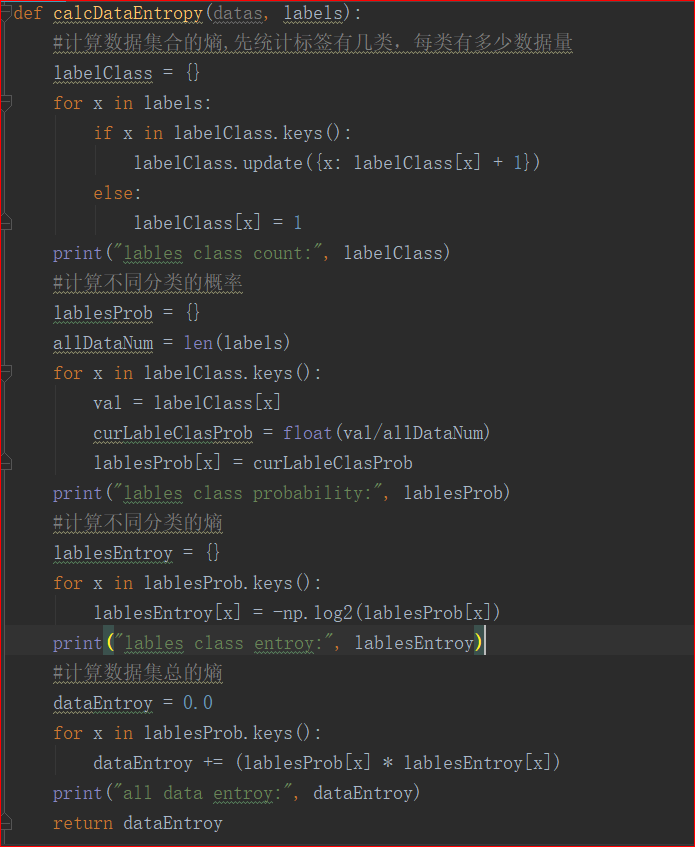
使用下面的海洋生物数据说明决策树的生成



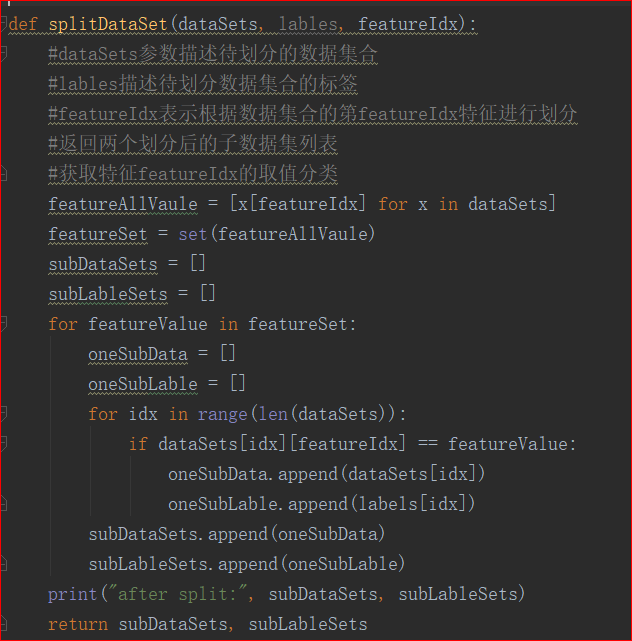
以上数据，在代码中我们用1表示是，用0表示否。用一个多维列表进行存储，代码描述该数据结构如下：



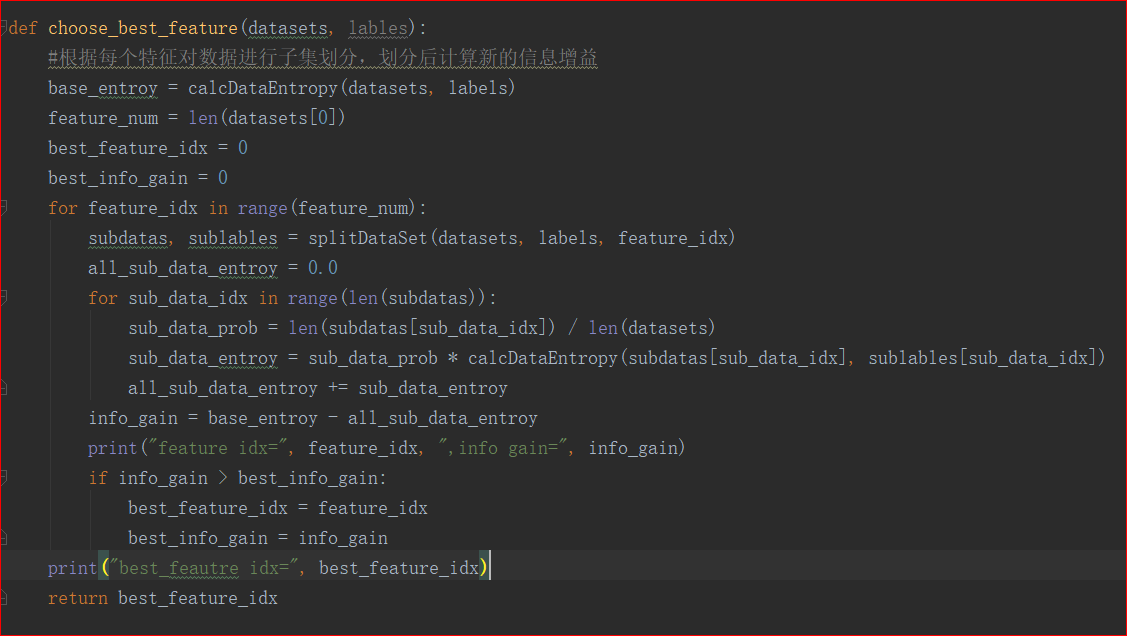
数据集的熵计算代码如下：



在构造决策树的过程中，需要根据数据的特征将数据集划分为多个子集



在构造决策树的过程中，选择一个特征时，需要确保信息丢失最小，如下函数实现数据集合上特征的选择：



最后我们完成决策数据的创建

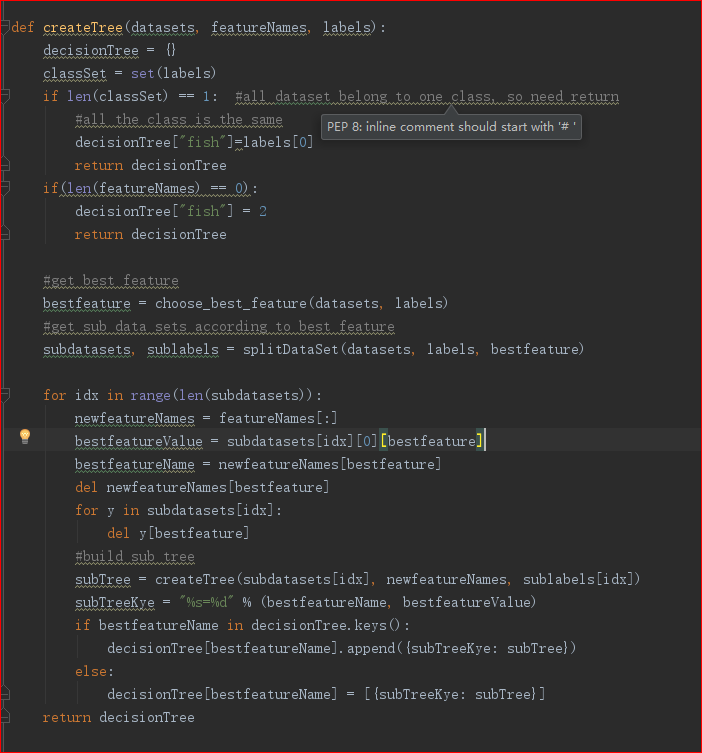
我们定义一个数据结构保存决策树的节点。在python中我们用字典作为决策树结点。

其中字典的key为决策判断所得到的特征，value是一个列表，为该特征不同取值下指向下一个节点。

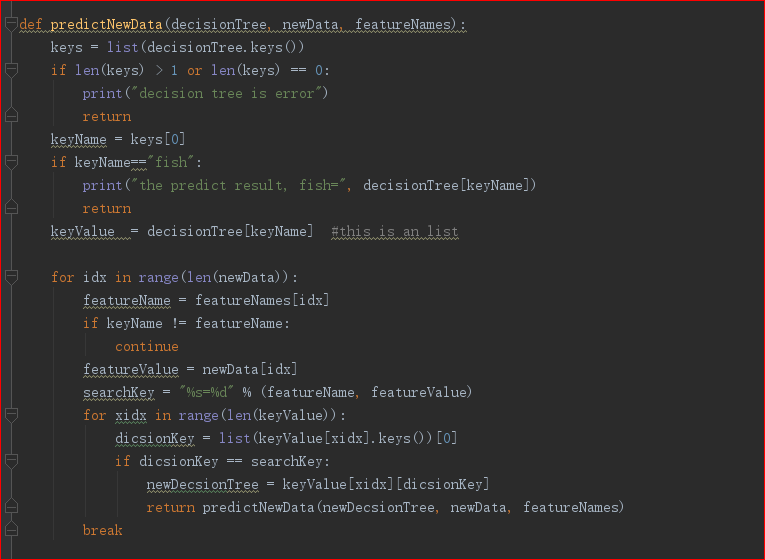
类似于如下结构：

{bestfeature:[{1:{bestfeature:[]}}, {2:{bestfeature[]}}, {3:{bestfeature:[]}}]

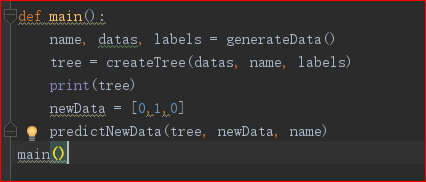
构造决策树代码如下（通过递归方式实现）



完成决策树以后，对于新的数据，则可以通过遍历决策数据，对新数据进行预测，预测代码流程如下：



整体代码的使用可以按照以下方式进行使用：



## 机器学习之朴素贝叶斯分类

学习贝叶斯分类算法之前，需要先掌握贝叶斯原理。

一些基本概念：

条件概率：在事件A发生的情况下，发生时间事件B的概率，一般用P(B|A)标记。

例如：

有一个红色箱子，一个白色箱子，在红色箱子中有3个黑色石头，4个灰色石头，在白色箱子中有4个黑色石头，有3个白色箱子。那么从红色箱子中去到黑色石头的概率是多少。类似于这样的问题被称为条件概率。

条件概率可以通过以下方式计算出结果

P(A|B) = P(A and B) P(B) 推导出 P（A and B） = P(A|B) / P(B)

B事件发生的情况下，A发生的概率 等价于 A和B同时发生 \* P单独发生的概率。

P(B|A) = P(B and A) P(A) 推导出 P(B and A) = P(B|A)/P(A)

根据以上两个推导，又因为P(A and B) = P(B and A)可以得到

P(B|A)/P(A) = P(A|B) / P(B)

P(B|A) =P(A|B) \* P(A)/P(B)

把公式P(B|A) =P(A|B) \* P(A)/P(B)我们称之为贝叶斯定理。换一种方式描述如下：

P(类别|特征) = P(特征|类别)\*P(类别)/P(特征)

利用贝叶斯定理，在机器学习中是如何进行分类了。

我们已经有一份数据集合，那么从该数据集合中，可以计算得到P(特征|类别)，P(类别)，P(特征)概率，然后通过该三者概率，可以计算新的数据属于某个类别的概率，从而进行分类。

该逻辑过程如下：

假定我们训练的数据集样本如下：



这时候有一个新的数据样本，内容如下：



那个根据新样本的各个特征取值，我们如何判断嫁或不嫁，将这个问题转换称贝叶斯定理，其过程如下：

1：计算不同分类下的概率

P(嫁|不帅，性格不好，矮，不上进) = P(不帅,性格不好，矮，不上进|嫁) \* P(嫁) / P(不帅，性格不好，矮，不上进)

P(不嫁|不帅，性格不好，矮，不上进) = P(不帅,性格不好，矮，不上进|不嫁) \* P(不嫁) / P(不帅，性格不好，矮，不上进)

在上面过程中：等式右边的内容都可以根据已有数据集合的内容计算得到。将等式展开如下：

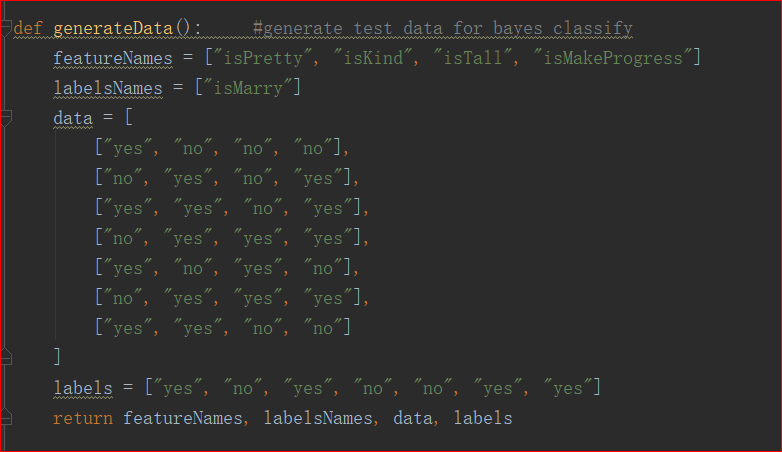
P(嫁|不帅，性格不好，矮，不上进) = [P(不帅|嫁) \* P(性格不好|嫁) \* P(矮|嫁) \* P(不上进|嫁) \* P(嫁)]/[P(不帅) \* P(性格不好) \* P(矮) \* P(不上进)]

P(不嫁|不帅，性格不好，矮，不上进) = [P(不帅|不嫁) \* P(性格不好|不嫁) \* P(矮|不嫁) \* P(不上进|不嫁) \* P(不嫁)]/[P(不帅) \* P(性格不好) \* P(矮) \* P(不上进)]

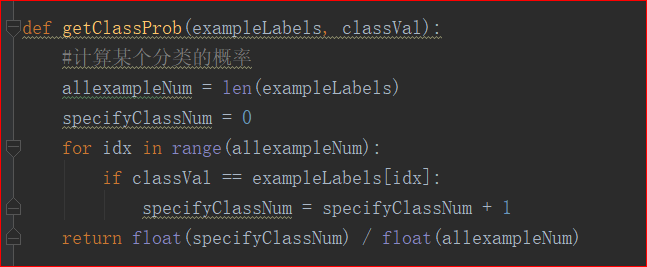
2：选择概率大的作为计算结果

代码如下：

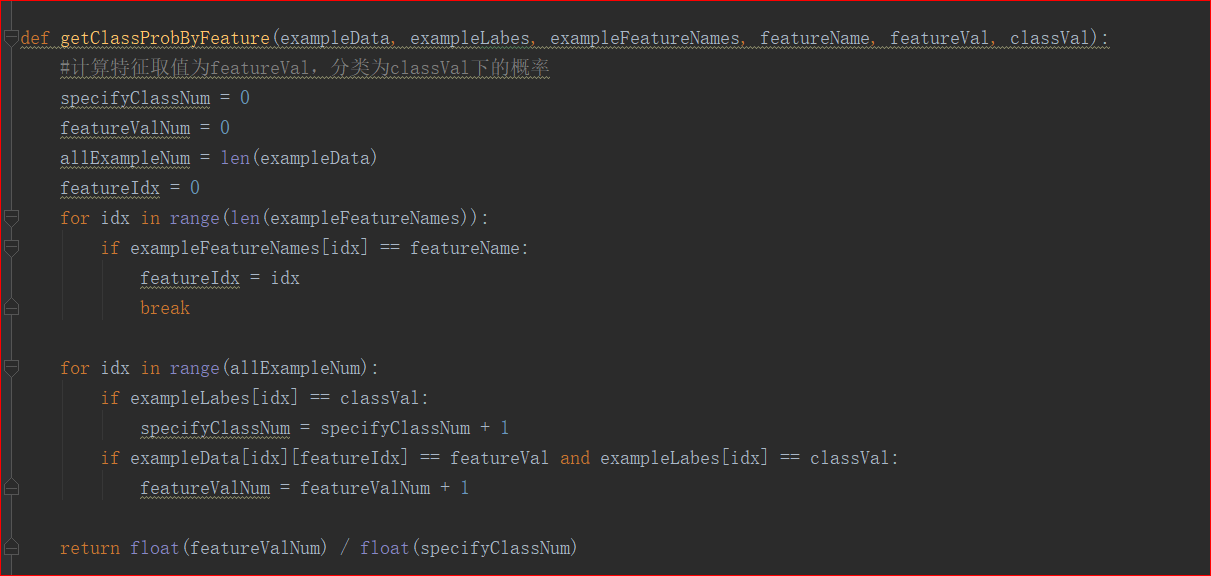
1：生成训练样例数据：



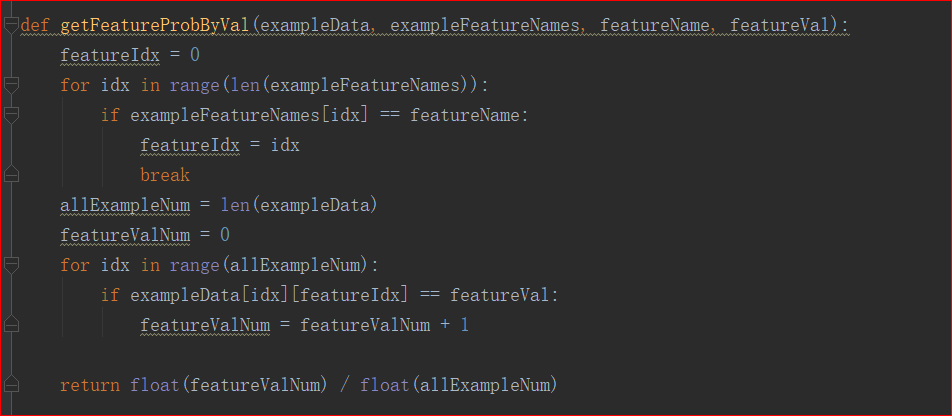
2：根据分类类型，计算在样例数据中的概率



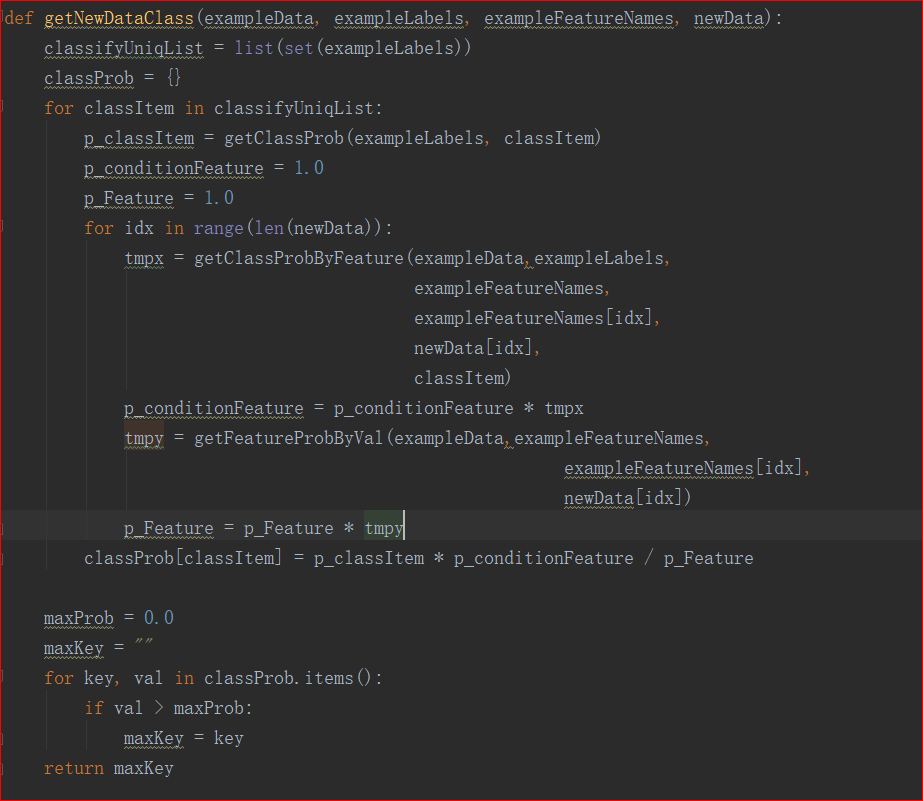
3：根据分类类型，特征取值为某一个特定只时的概率



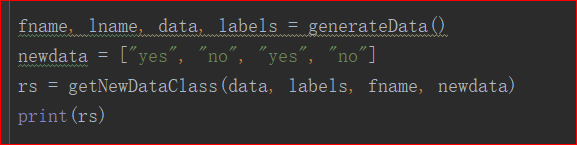
4：根据特征取值计算该特征取值下的概率



5：根据新的数据计算该数据在不同分类下的概率并作出判断



6：得到预测结果：



## 机器学习之SVM

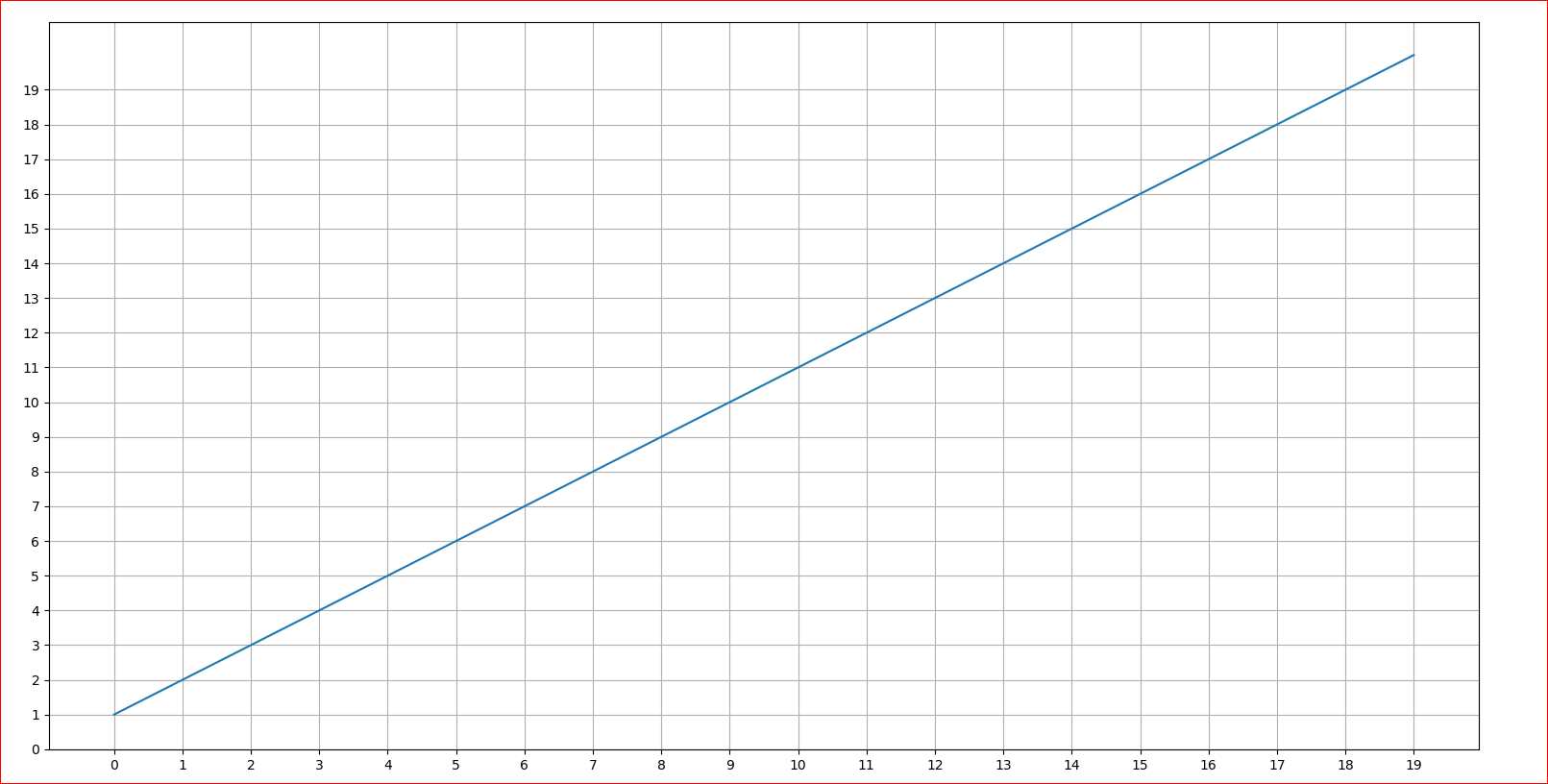
在进行数据的二元分类时，其实质是寻找一条直线或曲线对数据进行分割，事实上，符合条件的线可能会非常多，那么什么样的线段能最好的对数据进行分割，SVM算法就是找一组能够进行分割数据的直线中最有效的哪条直线（在这里，对数据的分割，我们用直线描述，事实上，如果数据超过3个维度，我们要寻找到的应该是一个超平面，也就是说是在找一个平面对数据进行分割，但其原理是相同的，所以后续我们一直使用直线代替）

SVM（支持向量机）算法就是找到一条直线，使的对需要分割的点而言，该直线到这些点的距离最大。

对于分离数据的直线，在数学上我们可以写为

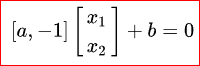
wTx + b = 0这样的形式。对于一个点A，到该直线的距离，可以写作|wTA + b|/||w||，下面举例说明：

对于直线 y=ax+b，我们假定a取值为1，b取值为1，那么该直线形式如下：



对该直线，做个变换，将x轴看作x1，y轴看作x2，格式上可以写成如下样子：

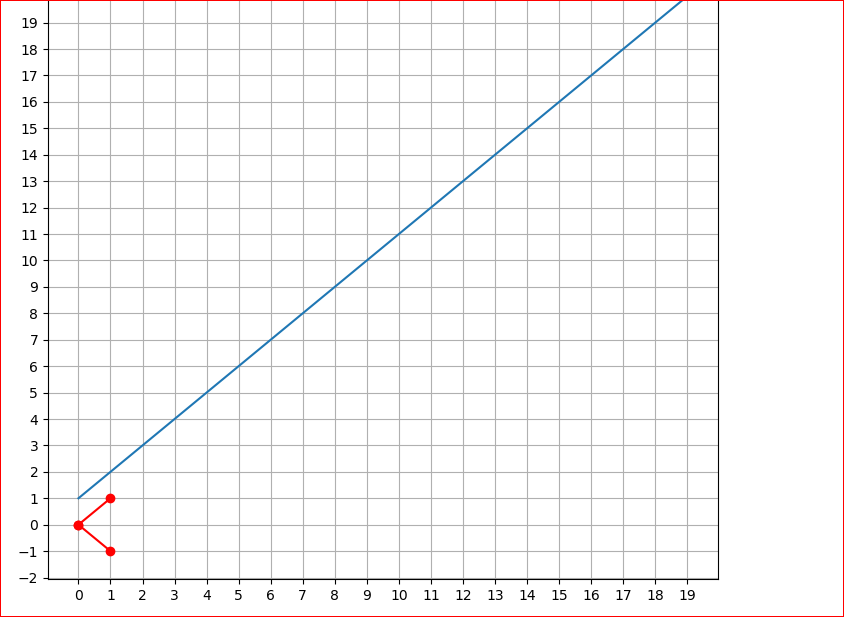
x2 = a\*x1+b，继续变换，可以写成a\*x1+(-1)\*x2+b = 0，继续变换写成向量形式：



将该矩阵写成更一般的形式，即令wTx+b = 0，其中

W是一个2行1列的向量，其形式如下：

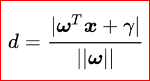
W的转置是一个1行2列的向量[a, -1]

在该过程中，a的取值是1，所以wT这个向量的取值是[1,-1]，此时如果我们在构造另外一个向量φ=[1,a]=[1,1]，我们会发现这wT与φ在坐标轴上互相垂直，其中φ与直线wTx+b = 0平行，图示如下：  


从以上图示可以看出，矩阵w控制了分割直线的方向，参数b控制了分割直线的截距（当横轴x取值为0时，分割直线y轴的取值）

对于二维平面，矩阵W是一个二维向量，对于多维，该参数是一个多维向量，可以类推。

对于已有数据，分布在该分割直线两侧的点，该如何求出该点到分割直线的距离了，在这里直接给出计算公式，对于一个点A，其到分割直线的距离如下：



例如：对于点A取值为[1,1]，该点到直线的距离为([1,-1]T \* [1, 1] + 1)/||[1, -1]|| = 1/

其中||w||称为向量的模，描述的是原点到该向量的距离。以上是基础概念。

下面说明SVM算法的求解过程：

对于已经存在的数据集合和存在的分割直线，那么在数据集合中肯定存在一些点，这些点到分割直线的距离总是小于其他点到分割直线的距离，也就是说不管分割直线如何变化，总有一些点到分割直线的距离小于集合中其他点到分割直线的距离。

对于SVM算法而言，这是求解的第一步，即找到数据集合中的一些点，这些点到分割直线的距离小于其他点到分割直线的距离。

当找到了以上这些点以后，在进行后续过程的求解，即找到一条分割直线，这些点到该分割直线的距离大于这些点到其他分割直线的距离。

完成以上两个过程以后，则SVM求解过程完成。

我们知道，在这个过程中。数据集是已知量，所要求解的分割线形式是已知量，但对应参数我们w和b是未知的，需要根据数据集合求解出参数w和b。

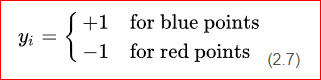
下面进行参数w和b的求解，求解之前会有一些约束条件：

1：并不是所有的分割平面都可以将样本数据进行分割。

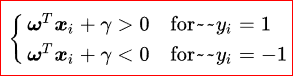
2：公式中的点并不是随随便便的一个样本点，而是支持向量对应的样本点

3：决策面的位置应该位于间隔区域的中轴线上。

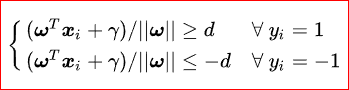
首先考虑一个决策面可以将所有样本点进行正确分类的约束。我们为每个样本点Xi加上也给类别标签Yi。

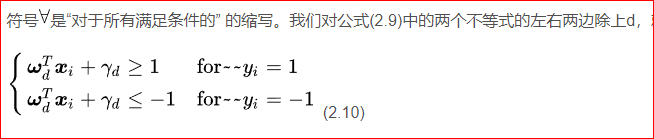


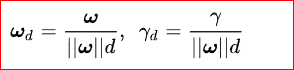
如果分割面能进行正确分类，那么则有如下表现：

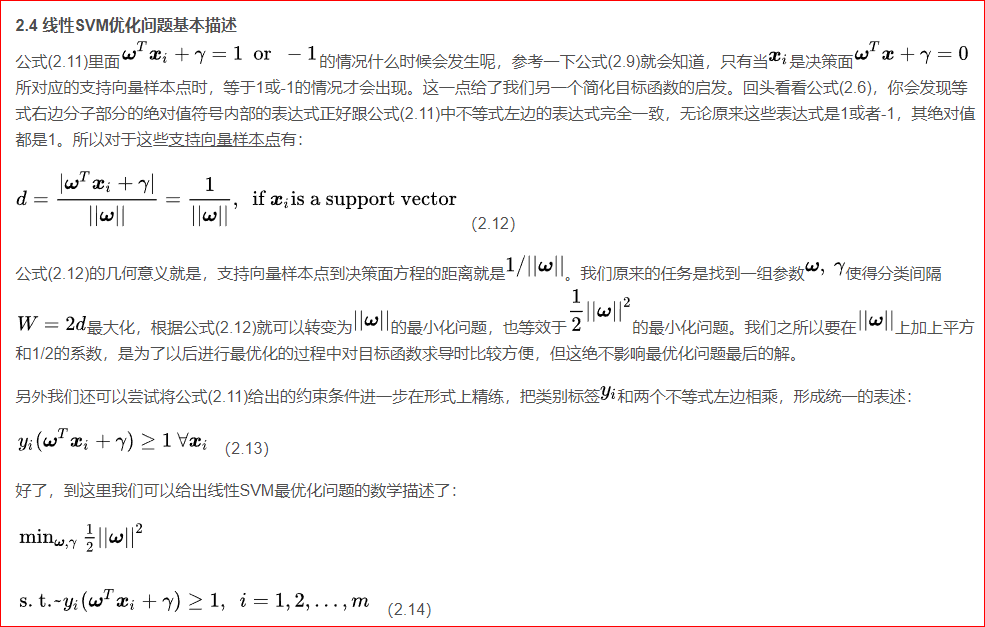


如果要求更高一点，假设分割平面位于中轴线上，并且响应向量对应的样本点到分割面的距离为d，则有









## Softmax实现mnist数据集合识别

import tensorflow as tf  
import numpy as np  
import struct  
import cv2 as cv

def inference\_with\_softmax(x\_input, w, b):  
 return tf.nn.softmax(tf.matmul(x\_input, w) + b)

def loss\_with\_softmax(x\_input, y\_input, w, b):  
 y\_predict = inference\_with\_softmax(x\_input, w, b)  
 y\_real = y\_input  
 cross\_entropy = -tf.reduce\_sum(y\_real \* tf.log(tf.clip\_by\_value(y\_predict, 1e-20, 1.0)))  
 return cross\_entropy

def train\_with\_softmax(total\_loss):  
 learning\_rate = 0.1  
 return tf.train.GradientDescentOptimizer(learning\_rate).minimize(total\_loss)

def eveulate\_with\_softmax(input\_x, input\_y, w, b):  
 y\_predict = inference\_with\_softmax(input\_x, w, b)  
 correct\_prediction = tf.equal(tf.argmax(y\_predict, 1), tf.argmax(input\_y, 1))  
 accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, "float"))  
 return accuracy

def get\_w\_b\_with\_sofmax(sess):  
 train\_data\_x, train\_data\_y = load\_data\_from\_mnist("E:\\mlDataSet\\mnist\\train-images.idx3-ubyte",  
 "E:\\mlDataSet\\mnist\\train-labels.idx1-ubyte")  
 train\_data\_x = tf.cast(train\_data\_x, tf.float32)  
 train\_data\_y = tf.cast(train\_data\_y, tf.float32)  
  
 w = tf.Variable(tf.zeros([784, 10]))  
 b = tf.Variable(tf.zeros([10]))  
 init = tf.global\_variables\_initializer()  
 sess.run(init)  
 loss = loss\_with\_softmax(train\_data\_x, train\_data\_y, w, b)  
 train = train\_with\_softmax(loss)  
 for i in range(1000):  
 print("Train step is:", i)  
 sess.run(train)  
 return w, b

def get\_evulate\_result(sess, w, b):  
 test\_img, test\_label = load\_data\_from\_mnist("E:\\mlDataSet\\mnist\\t10k-images.idx3-ubyte",  
 "E:\\mlDataSet\\mnist\\t10k-labels.idx1-ubyte")  
 test\_img = tf.cast(test\_img, tf.float32)  
 predict\_result = inference\_with\_softmax(test\_img, w, b)  
 y = sess.run(predict\_result)  
 same\_count = 0  
 all\_count = 0  
 for idx in range(len(y)):  
 all\_count = all\_count + 1  
 if (y[idx] == test\_label[idx]).all():  
 same\_count += 1  
 print("test rate = ", float(same\_count/all\_count))

def test\_main():  
 gpu\_options = tf.GPUOptions(per\_process\_gpu\_memory\_fraction=0.6)  
 config = tf.ConfigProto(gpu\_options=gpu\_options)  
 sess = tf.Session(config=config)  
 w, b = get\_w\_b\_with\_sofmax(sess)  
 get\_evulate\_result(sess, w, b)  
  
test\_main()

## CNN实现mnist数据集合识别

1：创建函数，从mnist原始文件中读取数据，对于28\*28的图片数据需要进行归一化操作，将数据全部分配到0到1的范围以内，具体读取代码如下：

def load\_data\_from\_mnist(dataname, labelname):  
 *"""return two matrix, one is mnist data, one is lables with one-host format"""* fp = open(dataname, "rb")  
 databuf = fp.read()  
 offsetidx = 0  
 magicNum, imageNum, imageRow, imageCol = struct.unpack\_from('>4I', databuf, 0)  
 offsetidx = struct.calcsize('>4I')  
  
 datasize = imageNum \* imageRow \* imageCol  
 readfmt = ">%dB" % datasize  
 imgdata = struct.unpack\_from(readfmt, databuf, offsetidx)  
 imgdata = np.array(imgdata)  
 imgdata = imgdata.astype(np.float32)  
 imgdata = imgdata.reshape(imageNum, imageRow \* imageCol)  
 imgdata = imgdata/255  
  
 fp.close()  
  
 fp = open(labelname, "rb")  
 labelbuf = fp.read()  
 offsetidx = 0  
 magicNum, imageNum = struct.unpack\_from('>2I', labelbuf, 0)  
 offsetidx = struct.calcsize('>2I')  
 readfmt = ">%dB" % imageNum  
 labeldata = struct.unpack\_from(readfmt, labelbuf, offsetidx)  
 labeldata = np.array(labeldata)  
 labeldata = labeldata.astype(np.int8)  
 labeldata = np.eye(10)[labeldata]  
 labeldata = labeldata.astype(np.float32)  
  
 fp.close()  
 return imgdata, labeldata

注意标红的位置，该行代码对数据进行归一化处理，如果不做归一化处理，后面训练结果将不正确，会导致训练参数中出现NA错误。

2：定义预测函数

预测函数的目的是根据现有的数据，计算出一个匹配结果，该函数的实现也是训练模型实现的关键部分。该函数内部可以划分为两部分

1. 所使用到的变量参数。
2. 数据之间的关系（模型关系）

def inference(input\_x, keep\_probe):  
 #define varible  
 w\_conv1 = weight\_variable([5, 5, 1, 32])  
 b\_conv1 = bias\_variable([32])  
 w\_conv2 = weight\_variable([5, 5, 32, 64])  
 b\_conv2 = bias\_variable([64])  
 w\_fc1 = weight\_variable([7 \* 7 \* 64, 1024])  
 b\_fc1 = bias\_variable([1024])  
 w\_fc2 = weight\_variable([1024, 10])  
 b\_fc2 = bias\_variable([10])  
  
 #defin model  
 x\_image = tf.reshape(input\_x, [-1, 28, 28, 1])  
 h\_cov1 = tf.nn.relu(conv2d(x\_image, w\_conv1) + b\_conv1)  
 h\_pool1 = max\_pool\_2x2(h\_cov1)  
 h\_cov2 = tf.nn.relu(conv2d(h\_pool1, w\_conv2) + b\_conv2)  
 h\_pool2 = max\_pool\_2x2(h\_cov2)  
 h\_pool2\_flat = tf.reshape(h\_pool2, [-1, 7 \* 7 \* 64])  
 h\_fc1 = tf.nn.relu(tf.matmul(h\_pool2\_flat, w\_fc1) + b\_fc1)  
 h\_fc1\_drop = tf.nn.dropout(h\_fc1, keep\_probe)  
 y\_conv = tf.nn.softmax(tf.matmul(h\_fc1\_drop, w\_fc2) + b\_fc2)  
 return y\_conv

注意预测函数中变量的定义，由于预测函数在主流程中会被调用多次，这样在其中定义的变量会被tensorflow框架定义多次，从而导致训练时所使用的变量与预测时所使用的变量并不是同一个变量，从而导致预测结果异常，因此在预测函数中变量的定义可以参考如下方式完成：

def weight\_variable(shape, name):  
 try:  
 with tf.variable\_scope("cnn", reuse=True):  
 var = tf.get\_variable(name=name, shape=shape)  
 return var  
 except ValueError:  
 with tf.variable\_scope("cnn", reuse=False):  
 var = tf.get\_variable(name=name, shape=shape)  
 return var

如上代码表示，在变量作用域cnn范围内，如果找到变量则使用该变量，如果没有找到，则创建该变量，从而确保预测函数中训练所使用的变量与预测所使用的变量是同一个变量。

2：定义损失函数

损失函数用于将预测结果与真实数据进行比较以便其差值达到最小。

def loss(input\_x, input\_y):  
 y\_predict = inference(input\_x, 0.5)  
 cross\_entropy = -tf.reduce\_sum(input\_y \* tf.log(y\_predict))  
 return cross\_entropy

3：定义训练函数

在tensorflow中使用对应的优化器进行训练

def train(total\_loss):  
 return tf.train.AdamOptimizer(1e-4).minimize(total\_loss)

4：定义训练主流程以及测试主流程

def main():  
 x\_input, y\_input = load\_train\_data()  
 data\_len = len(x\_input)  
 batch\_size = 50  
 batch\_num = data\_len / batch\_size  
  
 x = tf.placeholder('float', shape=[None, 784])  
 y = tf.placeholder('float', shape=[None, 10])  
 total\_loss = loss(x, y)  
 train\_op = train(total\_loss)  
 predict\_op = inference(x, 1.0)  
  
 sess = tf.Session()  
 sess.run(tf.global\_variables\_initializer())  
  
 for idx in range(2000):  
 batch\_idx = int(idx % batch\_num)  
 batch\_x = x\_input[batch\_idx \* batch\_size: (batch\_idx + 1) \* batch\_size]  
 batch\_y = y\_input[batch\_idx \* batch\_size: (batch\_idx + 1) \* batch\_size]  
  
 sess.run(train\_op, feed\_dict={x: batch\_x, y: batch\_y})  
 if idx % 100 == 0:  
 print("Train step is ", idx)  
  
 #在数据集合上使用预测函数，检查预测结果  
 tst\_x\_input, tst\_y\_input = load\_test\_data()  
 data\_len = len(x\_input)  
 same\_cnt = 0  
 all\_cnt = 0  
 for idx in range(data\_len):  
 batch\_x = tst\_x\_input[idx: idx + 1]  
 batch\_y = tst\_y\_input[idx: idx + 1]  
 rs = sess.run(predict\_op, feed\_dict={x: batch\_x, y: batch\_y})  
  
 rs = soft\_max\_result\_to\_one\_shot(rs)  
 if (rs == batch\_y).all():  
 same\_cnt = same\_cnt + 1  
 all\_cnt = all\_cnt + 1  
  
 print("accuracy = ", same\_cnt / all\_cnt)