《机器学习实战》源码解析

机器学习源码解析 I

Python 开发环境安装及机器学习基础

jackycaf

2014/5/3 Saturday

目录

第一章 Python 数学分析环境安装 3
Python 的安装： 3
numpy 的安装： 5
matplotlib 的安装： 6
Scipy 的安装： 7
UltraEdit 绿色版的安装及 Python 开发环境的配置 8
测试 Python 数学分析开发环境 10
第二章 Python Numpy 的矩阵编程 13
Numpy 矩阵常用函数与编程 13
Numpy 特殊矩阵 17
numpy linalg 库的矩阵运算 20
矩阵常用函数的实现 22
第三章 实现机器学习的各种距离 29
1. 欧氏距离(Euclidean Distance) 29
2. 曼哈顿距离(Manhattan Distance) 30
3. 切比雪夫距离(Chebyshev Distance) 31
4. 闵可夫斯基距离(Minkowski Distance) 31
5. 标准化欧氏距离(Standardized Euclidean distance) 32
6. 马氏距离(Mahalanobis Distance) 33
7. 夹角余弦(Cosine) 34
8. 汉明距离(Hamming distance) 35
9. 杰卡德相似系数(Jaccard similarity coefficient) 35
10. 相关系数(Correlation coefficient)与相关距离(Correlation distance) 36
11. 信息熵(Information Entropy) 37

第一章 Python 数学分析环境安装

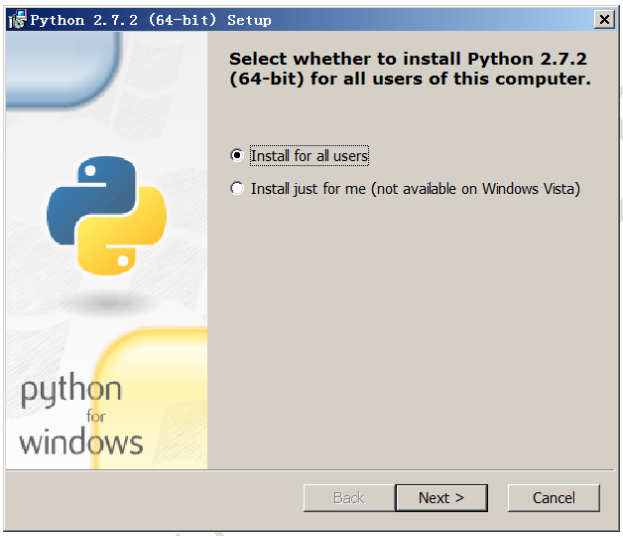
理解本文的前提是需要熟悉 Python 程序的开发。对 Python 不熟悉的朋友首先请观看“www.threedweb.cn”论坛的 Python 板块的视频教程，教程的帖子地址为：http://www.threedweb.cn/thread-41-1-1.html。

本文所述的所有安装包均可从论坛：www.threedweb.cn 的 python 板块的相关帖子上下载，下载地址：http://www.threedweb.cn/thread-42-1-1.html。

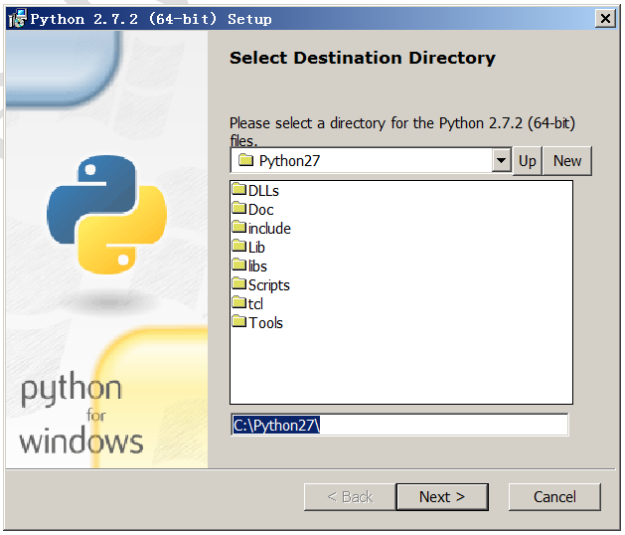
Python 的安装：

1. *下载安装包：python-2.7.2.exe*

2. *双击运行安装包：*

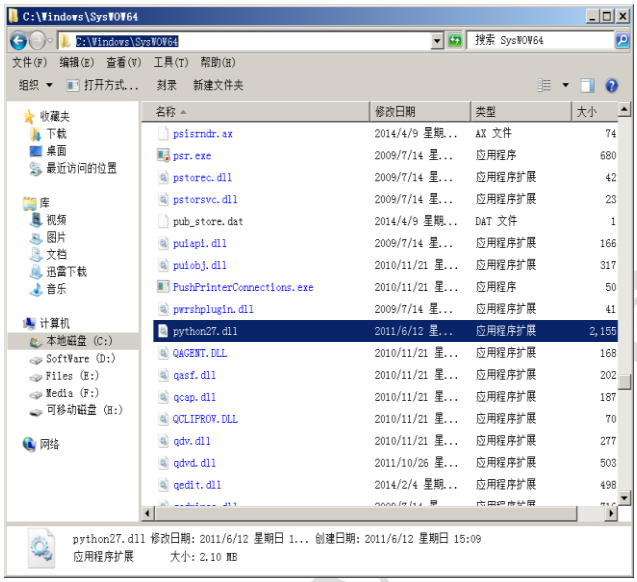


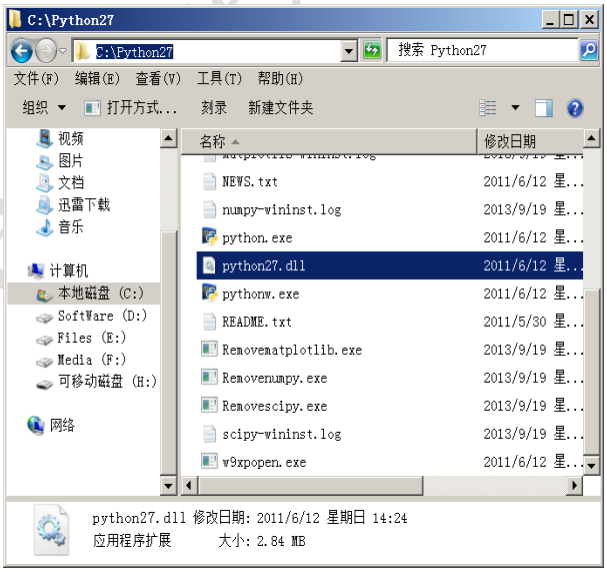
3. *指定安装目录: C:\Python27*



4. *之后一路 Next 直到安装完成。*

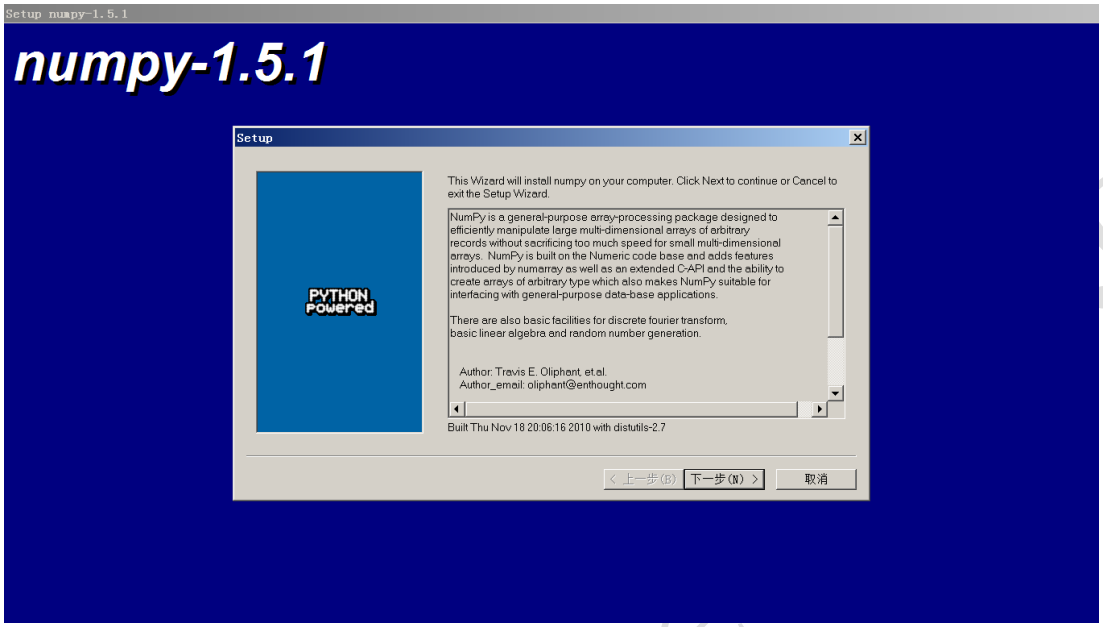
5. 制作绿色安装包：将 C:\Windows\ SysWOW64\python27.dll 拷贝到C:\Python27 安装目录下



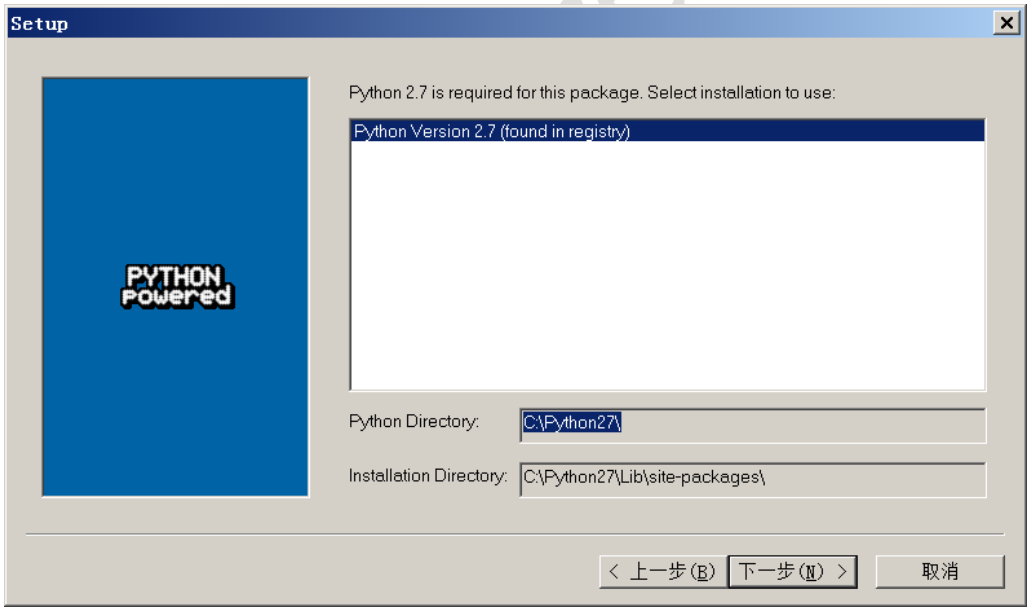


numpy 的安装：

运行 numpy-1.5.1-win32-superpack-python2.7.exe 安装包。



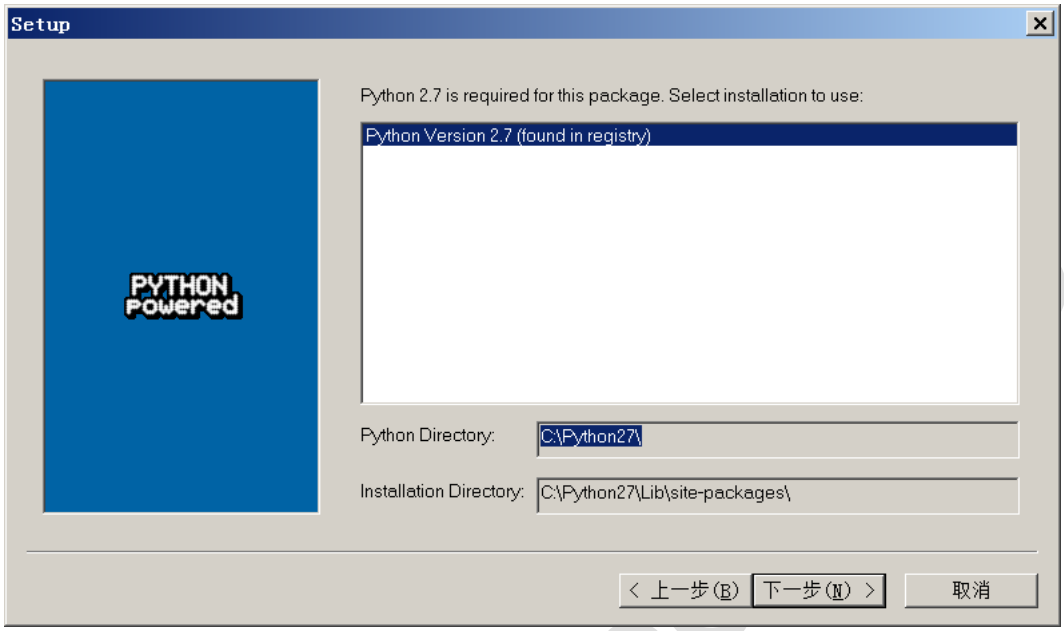
Numpy 会自动找到 python 的安装路径。



点击下一步安装。直至完成。

matplotlib 的安装：

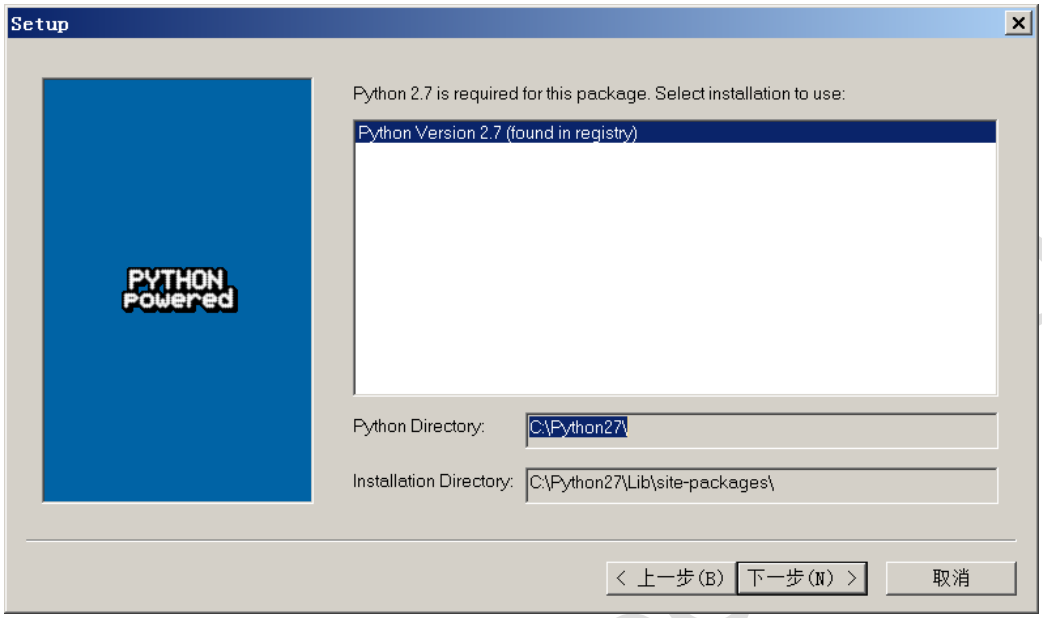
运行 matplotlib-1.1.0.win32-py2.7.exe 安装包。
matplotlib 会自动找到 python 的安装路径。



点击下一步安装。直至完成。

Scipy 的安装：

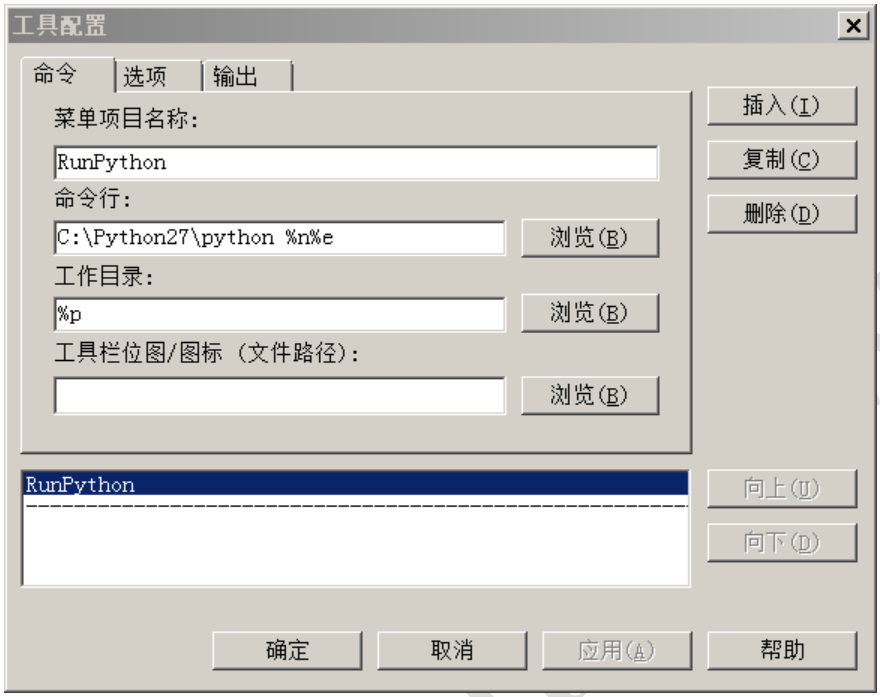
运行 scipy-0.12.0c1-win32-superpack-python2.7 安装包。
scipy 会自动找到 python 的安装路径。

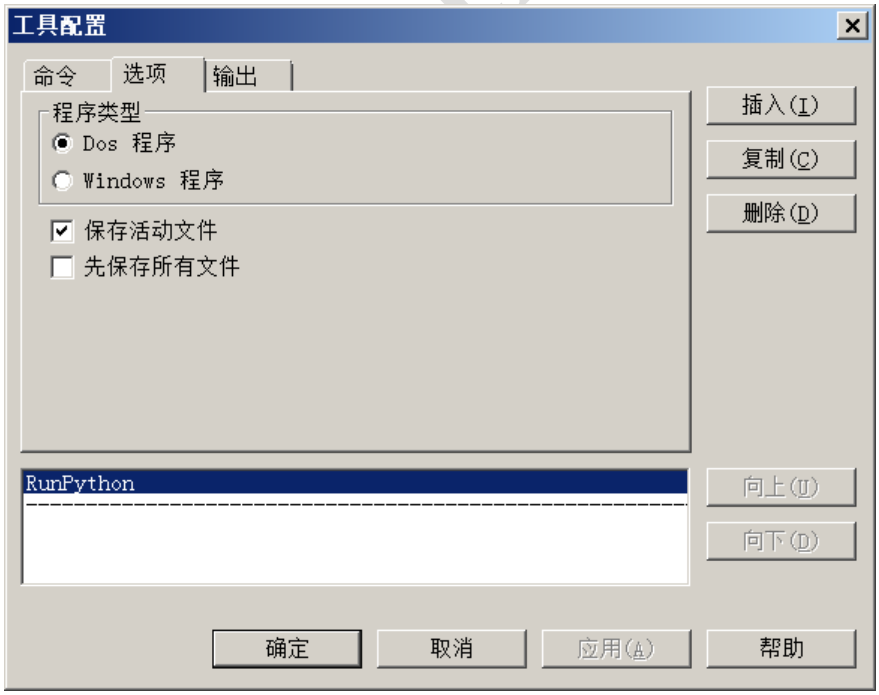


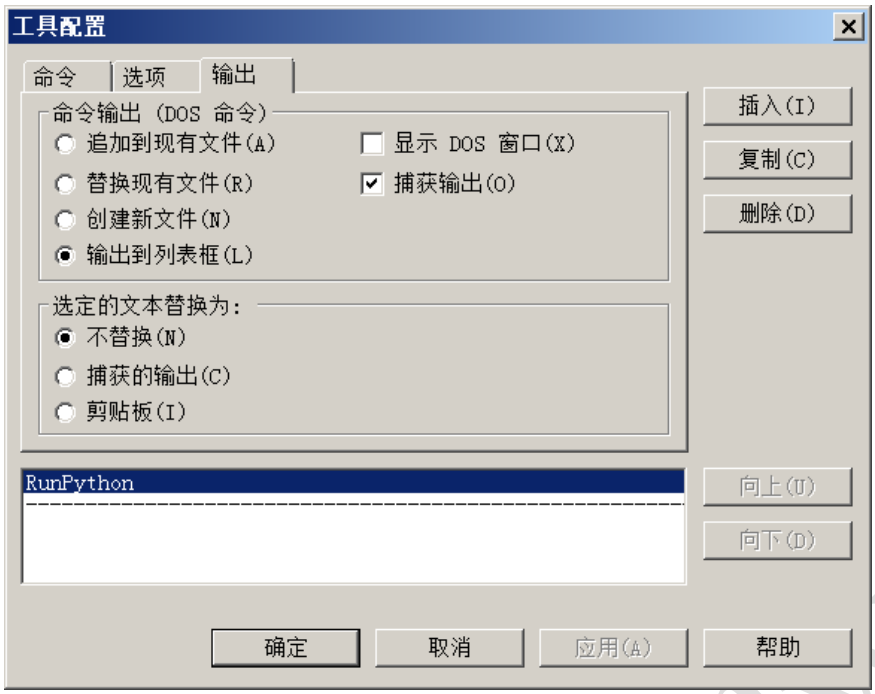
点击下一步安装。直至完成。

UltraEdit 绿色版的安装及 Python 开发环境的配置

解压 UltraEdit-开发环境.rar 文件到指定目录下，这里的安装目录是：D:\Program Files\UltraEdit。
运行 Uedit32.exe 文件，打开编辑器的界面。 在菜单栏选择“高级”->“工具栏配置”，点击“插入”按钮。
根据窗口信息按下图输入。



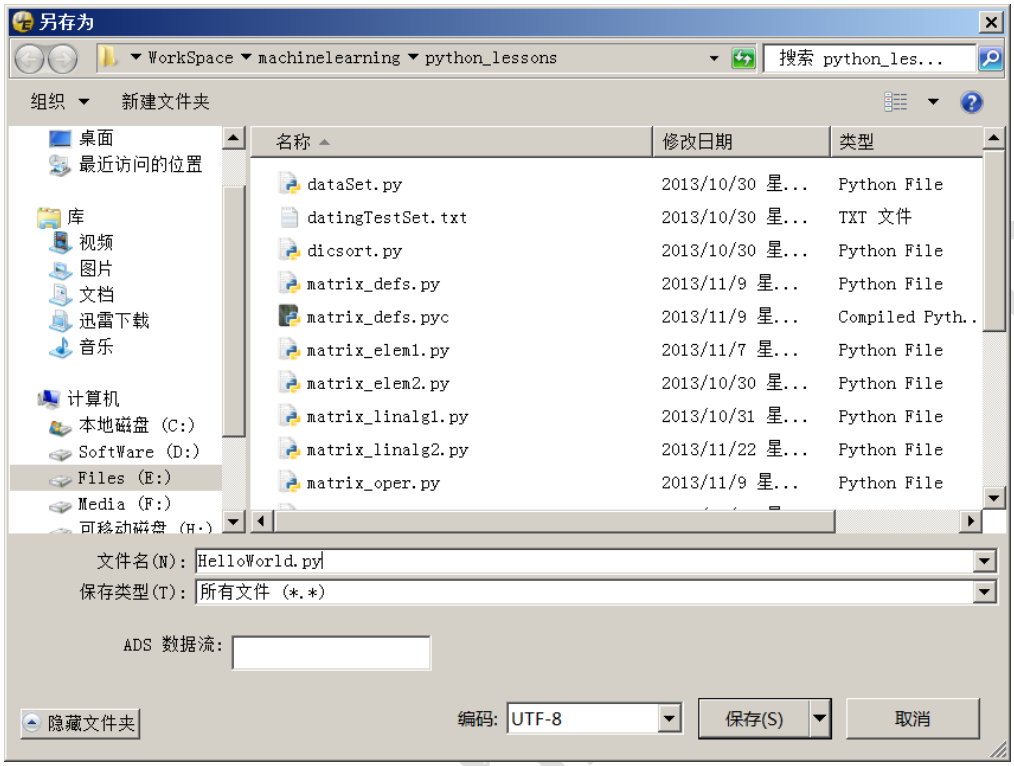




测试 Python 数学分析开发环境

1. 测试标准 python 程序

使用 UltraEdit 新建一个 helloworld.py 文件:



代码区：HelloWorld.Py

# -*- coding: utf-8 -*- 	
# Filename : helloworld.py 	
 	
# 中文输入 	
a = "世界 你好!"
b = "Hello World"
print a
print b
	# -*- coding: utf-8 -*- 表示输出格式为 utf-8

输出区：

```
世界 你好!
Hello World
```

2. 测试标准 python 数学分析包程序

本例程所使用的函数可参照 Numpy 和 Matplotlib 用户文档，可从论坛: www.threedweb.cn 的 python 板块的相关帖子下载，下载地址：

http://www.threedweb.cn/thread-36-1-1.html,

http://www.threedweb.cn/thread-40-1-1.html 。

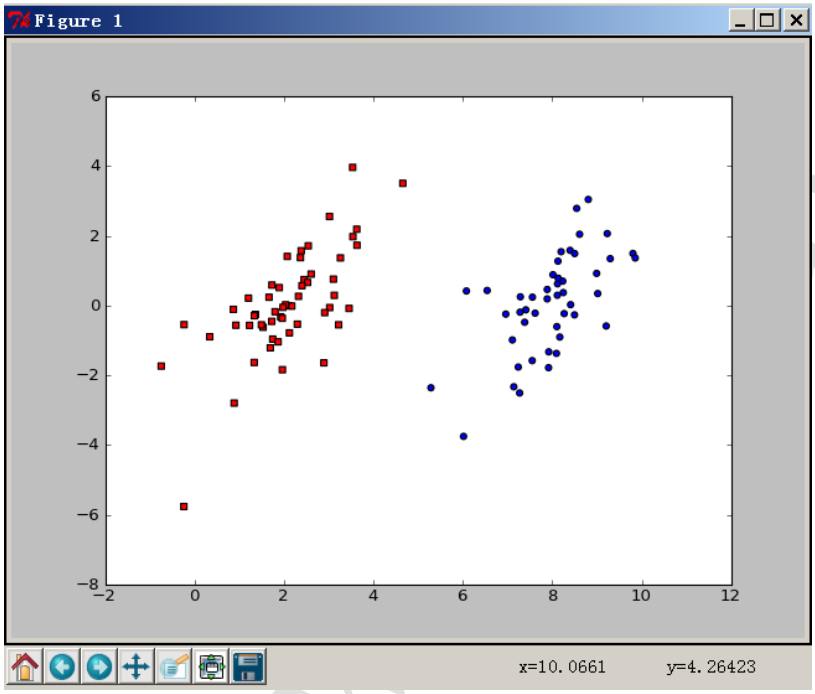
使用 UltraEdit 新建一个 dataSet.py 文件

代码区：dataSet.py

```
# -*- coding: utf-8 -*-
# Filename : dataSet.py

from numpy import *
import operator
import numpy as np
import matplotlib.pyplot as plt

#  文件 testSet.txt 中的数据集:
# 3.542485	1.977398	-1
# 3.018896	2.556416	-1
# 7.551510	-1.580030	1
# 2.114999	-0.004466	-1
# 8.127113	1.274372	 1
# 7.108772	-0.986906	1
# 8.610639	2.046708	1
# 2.326297	0.265213	-1
# 3.634009	1.730537	-1
# 0.341367	-0.894998	-1
# 3.125951	0.293251	-1
# 2.123252	-0.783563	-1
# 0.887835	-2.797792	-1
# 7.139979	-2.329896	1
# 1.696414	-1.212496	-1


# dataSet 导入
dataMat = []; classLabels = []
fr = open("testSet.txt")
for line in fr.readlines():
    lineArr = line.strip().split()
    dataMat.append([float(lineArr[0]), float(lineArr[1])])
```

```
        classLabels.append(int(lineArr[2]))

# 绘制图形
i = 0;
for mydata in dataMat:
        if classLabels[i]==1:
                plt.scatter(mydata[0],mydata[1],c='blue',marker='o')
        if classLabels[i]==-1:
                plt.scatter(mydata[0],mydata[1],c='red',marker='s')
        i += 1
plt.show()
```

输出图形为：



第二章 Python Numpy 的矩阵编程

Numpy 矩阵常用函数与编程

1. 矩阵的初始化：matrix01.py

```
# -*- coding: utf-8 -*-
# Filename : matrix01.py

from numpy import *
import operator
import matplotlib.pyplot as plot

# numpy 矩阵的初始化

# 使用 list 初始化矩阵
myArr = []
arr1 = [ 3.542485,1.977398,0 ]
arr2 = [ 3.018896,2.556416,1 ]
arr3 = [ 1,2,3 ]
arr4 = [ 7.551510,-1.580030,1]
# 使用 append 构成二维数组
myArr.append(arr1)
myArr.append(arr2)
myArr.append(arr4)
# 初始化 numpy 矩阵
mymat=mat(myArr)
print "myArr:",mymat
```

输出：

```
mymat: [[ 3.542485    1.977398    0.          ]
 [ 3.018896    2.556416    1.          ]
 [ 7.55151    -1.58003     1.          ]]
```

2. 矩阵的转置：

```
# 矩阵转置: numpy 的 matrix 函数
mymatT = transpose(mymat)
print "mymatT:",mymatT
print
```

输出：

```
mymatT: [[ 3.542485    3.018896    7.55151 ]
 [ 1.977398    2.556416 -1.58003 ]
 [ 0.          1.          1.        ]]
```

3. 矩阵的增加一列：

```
# 矩阵增加列: matrix->list->matrix
# 1.转换：tolist()
# 2.附加: append()
# 3.转置: transpose()
myArrT = mymatT.tolist()
myArrT.append(arr3)
mymat = transpose(myArrT)
print "mymat:",mymat
print
```

输出：

```
mymat: [[ 3.542485    1.977398    0.          1.        ]
 [ 3.018896    2.556416    1.          2.        ]
 [ 7.55151    -1.58003     1.          3.        ]]
```

4. 矩阵的删除一列或一行：

```
# 矩阵删除行:list 层
myArr1 = mymat.tolist()
myArr1.pop(1) # del myArr1[1]
mymat2 = mat(myArr1)
print "mymat:",mymat2

# 矩阵删除列:list 层
mymatT = transpose(mymat)
myArr2 = mymatT.tolist()
myArr2.pop(1)
mymat = transpose(myArr2)
print "mymat:",mymat
```

输出：

```
mymat: [[ 3.542485    1.977398    0.          1.        ]
 [ 7.55151    -1.58003     1.          3.        ]]

mymat: [[ 3.542485    0.          1.        ]
 [ 3.018896    1.          2.        ]
 [ 7.55151     1.          3.        ]]
```

5. 获取矩阵的列与行：

```
# 获取矩阵的行列
# shape(): 获取行、列--list,matrix 层
[m,n]=shape(mymat)
print m,n
print
```

输出：

```
3 3
```

6. 矩阵的遍历：

```
# 简单遍历矩阵:第 2 列
print "mymat[:,1]",mymat[:,1]

# 简单遍历矩阵:第 2 行
print "mymat[1,:]",mymat[1,:]
print

# 循环遍历 1-行遍历:list 层
i = 0
for elem in mymat:
    print "mymat[%s,:]=%s,%s,%s"%(i,elem[0],elem[1],elem[2])
    i = i+1;
print

# 循环遍历 2-列遍历:n 为矩阵的列数
for col in range(n):
    print "mymat[:,%s]=%s,%s,%s"%(col,mymat[0,col],mymat[1,col],mymat[2,col])
```

输出：

```
mymat[:,1] [ 0.    1.    1.]
mymat[1,:] [ 3.018896    1.              2.          ]

mymat[0,:]=3.542485,0.0,1.0
mymat[1,:]=3.018896,1.0,2.0
mymat[2,:]=7.55151,1.0,3.0

mymat[:,0]=3.542485,3.018896,7.55151
mymat[:,1]=0.0,1.0,1.0
mymat[:,2]=1.0,2.0,3.0
```

7. 矩阵的操作：分割，复制，排序

```
# 矩阵的分割
col1 = mymat[:,1]          # 按列分割
raw1 = mymat[1,:]      # 按行分割
print "col1:",col1
print "raw1:",raw1
print
# 矩阵的复制:深层复制 --matrix 层
mymatcp = mymat.copy() # 转换为 numpy 矩阵形式,进行深层复制
print "mymatcp:",mymatcp
print
# 矩阵的排序和逆序--list 层
matarr = mymatcp.tolist()[1]
matarr.sort()
print matarr
print
```

输出：

```
col1: [ 0.    1.    1.]
raw1: [ 3.018896    1.            2.          ]

mymatcp: [[ 3.542485    0.            1.          ]
 [ 3.018896    1.            2.          ]
 [ 7.55151     1.            3.          ]]

[1.0, 2.0, 3.018896]

[3.018896, 2.0, 1.0]
```

8. 矩阵的操作：分割，复制，排序

```
# 检验数据类型
print "type(matarr):",type(matarr)     # list
print "type(mymat):",type(mymat)     # matrix
print
# 判断列表相等
A= [1,2,3];B= [1,2,3]
print "A == B:", A == B
print "A is B:", A is B
print
# 判断矩阵的大小
A = mat([1,2,3]); B = mat([4,5,6])
print "A<B:",A<B
```

```
print
# 返回矩阵最大的元素
print "max:",A.max()," min:",B.min()
print
```

输出：

```
type(matarr): <type 'list'>
type(mymat): <type 'numpy.ndarray'>

A == B: True
A is B: False

A<B: [[ True  True  True]]

max: 3  min: 4
```

Numpy 特殊矩阵

1. 全零矩阵：matrix02.py

```
# -*- coding: utf-8 -*-
# Filename : matrix02.py

from numpy import *
import operator
import matplotlib.pyplot as plot

# 特殊的矩阵
# 全零矩阵：3 行 4 列全 0 矩阵
m3_4 = [3,4]
myZero = zeros(m3_4)
print "myZero:",myZero
print
```

输出：

```
myZero: [[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

2. 全 1 矩阵：

```
# 全 1 矩阵
```

```
myOnes = ones(m3_4)
print "myOnes:",myOnes
print
```

输出：

```
myOnes: [[ 1.   1.   1.   1.]
 [ 1.   1.   1.   1.]
 [ 1.   1.   1.   1.]]
```

3. 随机矩阵：

```
# 随机矩阵:3 行 4 列的 0~1 之间的随机数矩阵
myRand = random.rand(3,4)
print "myRand:",myRand
print
```

输出：

```
myRand: [[ 0.52685717   0.3834608    0.23860809   0.00778079]
 [ 0.56625964   0.84239684   0.75666807   0.33389922]
 [ 0.43823756   0.66823766   0.65581204   0.77825111]]
```

4. 单位阵：

```
# 单位阵
myEye = eye(3)
print "myEye:",myEye
print
```

输出：

```
myEye: [[ 1.   0.   0.]
 [ 0.   1.   0.]
 [ 0.   0.   1.]]
```

5. 元素级运算：同行同列

```
myMat1 = ones(m3_4)
myMat2 = random.rand(3,4)
M1 =   myMat1+myMat2      # 元素相加
M2 =   myMat1-myMat2      # 元素相减
print "M1:",M1
print "M2:",M2
print
```

输出：

```
M1: [[ 1.76283411  1.18266412  1.01722116  1.93663154]
 [ 1.9979826   1.35567246  1.08337109  1.35015104]
 [ 1.15121344  1.00933964  1.97205746  1.1908906 ]]
M2: [[ 0.23716589  0.81733588  0.98277884  0.06336846]
 [ 0.0020174   0.64432754  0.91662891  0.64984896]
 [ 0.84878656  0.99066036  0.02794254  0.8091094 ]]
```

6. 元素级运算：所有元素的和

```
smmyMat1 = sum(myMat1)
print "smmyMat1:",smmyMat1
print
```

输出：

```
smmyMat1: 12.0
```

7. 矩阵各元素的积：点积 (.*)

```
# 矩阵各元素的积 matlab .*
A=[[1,2,3],[2,3,4],[4,5,6]];B=[[9,8,7],[6,5,4],[3,4,5]]
AB = multiply(A,B)
print "multiply(A,B):",AB
print
```

输出：

```
multiply(A,B): [[ 9 16 21]
 [12 15 16]
 [12 20 30]]
```

8. 矩阵各元素的 n=2 次幂

```
pwM2 = power(M2,2)
print "pwM2:",pwM2 # 或 M2**2
print
```

输出：

```
pwM2: [[  5.62476596e-02   6.68037944e-01   9.65854243e-01   4.01556119e-03]
 [  4.06988815e-06   4.15157977e-01   8.40208552e-01   4.22303674e-01]
 [  7.20438632e-01   9.81407958e-01   7.80785511e-04   6.54658015e-01]]
```

9. 矩阵各元素的开方:

```
sqM2 = sqrt(pwM2)
```

```
print "sqM2:",sqM2
print
```

输出：

```
sqM2: [[ 0.23716589   0.81733588   0.98277884   0.06336846]
 [ 0.0020174    0.64432754   0.91662891   0.64984896]
 [ 0.84878656   0.99066036   0.02794254   0.8091094 ]]
```

10. 矩阵的乘法：将 list 转换为 mat，此时 inner(A,B), dot(A,B) =A*B:

```
# 矩阵的乘法:将列表转换为 mat，此时 inner(A,B),dot(A,B) =A*B
A = [[1,2,3],[4,5,6],[7,8,9]];B = [[1],[4],[7]]
A=mat(A);B=mat(B)      # 转换为二维矩阵
AB = dot(A,B)          #计算 A,B 的点积
print "dot(A,B):",AB
print "A*B=",A*B
```

输出：

```
dot(A,B): [[ 30]
 [ 66]
 [102]]
A*B= [[ 30]
 [ 66]
 [102]]
```

numpy linalg 库的矩阵运算

这部分要调用 numpy 的 linalg 库（线性代数库）可以满足大多数的线性代数运算。

1. 矩阵的行列式：

```
# -*- coding: utf-8 -*-
# Filename : matrix04.py

from numpy import *
import numpy.linalg     # numpy 的 linalg 库
import operator
import matrix_defs
import matplotlib.pyplot as plot

# n 阶方阵的行列式运算
A = [[1,2,4,5,7,],[9,12,11,8,2,],[6,4,3,2,1,],[9,1,3,4,5],[0,2,3,4,1]]
```

```
detA =linalg.det(A);   # 方阵的行列式
print "linalg.det(A):",detA
```

输出：

```
linalg.det(A): -812.0
```

2. 矩阵的转置：

```
# 矩阵的转置
A = mat([[1,2,3],[4,5,6],[7,8,9]])
AT = A.T
print "A.T",AT
print
```

输出：

```
A.T [[1 4 7]
 [2 5 8]
 [3 6 9]]
```

3. 矩阵的逆：

```
# 矩阵的逆: linalg.inv()
A = [[8,1,6],[3,5,7],[4,9,2]]
invA = linalg.inv(A)
print "linalg.inv(A):",invA
print
```

输出：

```
linalg.inv(A): [[ 0.14722222 -0.14444444  0.06388889]
 [-0.06111111  0.02222222  0.10555556]
 [-0.01944444  0.18888889 -0.10277778]]
```

4. 生成对称矩阵：

```
# 产生对称矩阵
A = mat([[8,1,6],[3,5,7]])
AT = A.T
SYMA = A*AT
print "矩阵的对称:",SYMA
print "对称矩阵的逆矩阵:",linalg.inv(SYMA)
print
```

输出：

```
矩阵的对称: [[101  71]
 [ 71  83]]
```

```
对称矩阵的逆矩阵: [[ 0.02483543 -0.02124476]
 [-0.02124476  0.03022142]]
```

5. 矩阵的秩：

```
# 矩阵的秩:自定义函数
A = [[1,2,2,2],[2,4,6,8],[3,6,8,10]]
# 矩阵求秩
rank = mat(A).ndim
print "mat(A).ndim:",rank
```

输出：

```
mat(A).ndim: 2
```

6. 齐次和非齐次线性方程组的解：1:方阵且可逆

```
# 齐次和非齐次线性方程组的解 1:方阵且可逆
A = [[8,1,6],[3,5,7],[4,9,2]] #x1,x2,x3
b = [1,0,0] # 解矩阵
S = linalg.solve(A,transpose(b))
print "linalg.solve(A,b):",S
print
```

输出：

```
linalg.solve(A,b): [ 0.14722222 -0.06111111 -0.01944444]
```

7. 齐次和非齐次线性方程组的解：2.使用逆矩阵求方程组的解

```
# 使用逆矩阵求方程组的解
S = (matrix_defs.inverse(mat(A))*mat(b).transpose())
print "(matrix_defs.inverse(mat(A))*mat(b).transpose()):",S
print
```

输出：

```
(matrix_defs.inverse(mat(A))*mat(b).transpose()): [[ 0.14722222]
 [-0.06111111]
 [-0.01944444]]
```

矩阵常用函数的实现

机器学习中的一些算法，如神经网络，PCA 等需要大量使用线性代数中的一些算法，诸如，矩阵的运算，方阵的行列式，矩阵化行最简型、特征向量和特征值等。在论坛的 http://www.threedweb.cn/thread-126-1-1.html 帖子中，可以下载到这些函数的 C++语言实现（代码和书籍）。

这里为了讲解方便，我们使用 Python 简单地实现了一版以上涉及的相关函数，仅供参考。

1. 函数库文件：

```
# -*- coding: utf-8 -*-
# Filename : matrix_defs.py

from numpy import *
import numpy.linalg

# 矩阵的转置
def transposeMat(matA):
    m,n = shape(matA)
    rtnMat = zeros((n,m))
    for i in range(n):
        for j in range(m):
            rtnMat[i,j] = matA[j,i]
    return rtnMat
```

```
# 矩阵的行列式
def detMat(matA):
    s = 0;
    m,n=shape(matA)
    if m !=n :
        print "m,n not equal"
        return
    if 0 == n: return 0;
    if 1 == n: return   matA[0,0];      #/*阶为1，按照定义计算*/
    if 2 == n: return   matA[0,0]*matA[1,1]-matA[0,1]*matA[1,0];

    lenth = n - 1;#子行列式的阶
    #/*按照定义，初始化一个子行列式数组的空间*/
    p = zeros((lenth,lenth))
    for k in range(n):
        for i in range(lenth):
            for j in range(lenth):
                if (i < k) :
                    p[i,j] = matA[i,(j + 1)]; # 初始化子行列式的值
                if (i >= k):
                    p[i,j] = matA[(i + 1),(j + 1)];
        s += power(-1, k) * matA[k,0] * detMat(p);# 递归计算
    return s;
```

```
# 矩阵的逆
def inverse(Amat):
    m,T = shape(Amat)
    if m != T :
```

```
        print "不是方阵,无逆"
        return
    if detMat(Amat)==0 :
        print "行列式为 0,无逆"
        return
    matA = mat(zeros((T,2*T)))
    matA[:,0:T] = Amat[:,0:T]
    # 构造中间矩阵
    invA = matA.copy()
    invA[:,T:2*T] = eye(T)[:,0:T]
    for i in range(T):
        for k in range(T):
            if (k != i):
                t = invA[k,i] / invA[i,i];
                for j in range(2*T):
                    x = invA[i,j] * t;
                    invA[k,j] = invA[k,j] - x;
    i = 0
    for inv in invA:
        invA[i,:] = inv/invA[i,i]
        i += 1
    matA = invA[:,T:2*T]
    return matA
```

```
# 矩阵乘法
def multiMat(matA,matB):
    ma,na = shape(matA)
    mb,nb = shape(matB)
    dotmat = mat(zeros((ma,nb)))
    # 判断是否能相乘
    if na != mb :
        print ("%s 不等于 %s,两矩阵不能相乘!"%(na,mb))
        return
    for i in range(ma) :
        for j in range(nb) :
            msum = 0
            for k in range(mb) :
                msum += matA[i,k]*matB[k,j]
                dotmat[i,j] = msum
    return dotmat
```

```
# 计算向量的点积
def dotVect(matA,matB):
```

```
    vectA = matA.tolist()[0]
    vectB = matB.tolist()[0]
    vSum = 0
    dim = len(vectA)
    if dim != len(vectB):
        print ("向量不同维，或不是向量，不能相乘!")
        return
    for i in range(dim):
        vSum += vectA[i]*vectB[i]
    return vSum
```

```
# 向量乘法
def vectMulti(matA,matB):
    ma,na = shape(matA)
    mb,nb = shape(matB)
    dotvect = mat(zeros((ma,nb)))
    if na != 1 and mb != 1 :
        print ("%s,%s 不等于 1，两向量不能相乘!"%(na,mb))
        return
    j = 0
    for i in range(ma):
        for j in range(nb):
            dotvect[i,j] = matA[i,0]*matB[0,j]
    return dotvect
```

```
# 矩阵相乘
def smartMulti(matA,matB):
    ma,na = shape(matA)
    mb,nb = shape(matB)
    dotmat = mat(zeros((ma,nb)))
    # 判断是否能相乘
    if na != mb :
        print ("%s 不等于 %s,两矩阵不能相乘!"%(na,mb))
        return
    for i in range(na) :
        dotmat += vectMulti(matA[:,i],matB[i,:])
    return dotmat
```

```
# 矩阵求秩
def rank(matA) :
    m,n = shape(matA);
    nn=m;
```

```
    a = matA
    if (m>=n): nn=n;
    k=0;
    for l in range(0,nn-1):
        q = 0.0;
        for i in range(1,m-1):
            for j in range(1,n-1):
                d = abs(a[i,j]);
                if (d > q):
                    q = d;
                    si = i;
                    js = j;
        if (q + 1.0 == 1.0): break;
        k = k + 1;
        if (si != l):
            for j in range(l,n-1):
                d = a[l,j];
                a[l,j] = a[si,j];
                a[si,j] = d;

        if (js != l) :
            for i in range(m-1):
                d = a[i,js];
                a[i,js] = a[i,l];
                a[i,l] = d;

        for   i in range(l+1,n-1):
            d = a[i,l] / a[l,l];
            for i in range(l+1,n-1):
                a[i,j] = a[i,j] - d * a[l,j];
    return k
```

```
# 行最简形--阶梯矩阵
def rref(matA):
    lead = 0;
    rowCount, columnCount = shape(matA)
    for r in range(rowCount):
        if (columnCount <= lead):
            break;
        i = r;
        while (matA[i, lead] == 0):
            i +=1 ;
```

```
            if (i == rowCount):
                i = r;
                lead +=1 ;
                if (columnCount == lead):
                    lead -= 1
                    break;
        for j in range(columnCount):
            temp = matA[r, j]
            matA[r, j] = matA[i, j]
            matA[i, j] = temp

        div = matA[r, lead];
        if div ==0 : div =1.0e-10
        for j in range(0,columnCount):
            matA[r, j] /= div;
        for j in range(rowCount):
            if (j != r):
                sub = matA[j, lead];
                for k in range(columnCount):
                    matA[j, k] -= (sub * matA[r, k]);
        lead +=1;
    return matA
```

```
# 行最简形-另一个算法
def ToReducedRowEchelonForm(M):
    if not M: return
    lead = 0
    rowCount = len(M)
    columnCount = len(M[0])
    for r in range(rowCount):
        if lead >= columnCount:
            return
        i = r
        while M[i][lead] == 0:
            i += 1
            if i == rowCount:
                i = r
                lead += 1
                if columnCount == lead:
                    return
        M[i],M[r] = M[r],M[i]
        lv = M[r][lead]
```

```
        M[r] = [ mrx / lv for mrx in M[r]]
        for i in range(rowCount):
            if i != r:
                lv = M[i][lead]
                M[i] = [ iv - lv*rv for rv,iv in zip(M[r],M[i])]
        lead += 1
    return M
```

```
# 施密特正交法:gram_schmidt(A)
def gram_schmidt(A):
    Num=0;#迭代步数
    Ahang,Alie = shape(A)    #矩阵的行和列
    dim = max(Ahang,Alie)
    v = mat(zeros((Ahang,Alie)));
    h = mat(zeros((Ahang,Alie)));
    v[:,0] = A[:,0] / linalg.norm(A[:,0])
    for j in range(1,Alie):
        for i in range(0,j):
            vsum = mat(zeros((Ahang,Alie))) #临时变量 存储后面的求和
            for k in range(0,j):
                h[k,j] = A[:,j].transpose()*v[:,k]
                vsum[:,0] = vsum[:,0]+float(h[k,j])*v[:,k]
                Num = Num+1
            v[:,j] = A[:,j] - vsum[:,0]
            v[:,j] = v[:,j] / linalg.norm(v[:,j]);
    # print "Num:",Num
    return v
```

第三章 实现机器学习的各种距离

在做机器学习分类时常常需要估算不同样本之间的相似性度量(Similarity Measurement)，这时通常采用的方法就是计算样本间的“距离”(Distance)。采用什么样的方法计算距离是很讲究，甚至关系到分类的正确与否。

本章的目的就是对常用的相似性度量作一个总结。并给出 python 的实现。

1. 欧氏距离(Euclidean Distance)

欧氏距离是最易于理解的一种距离计算方法，源自欧氏空间中两点间的距离公式。

(1)二维平面上两点 a(x1,y1)与 b(x2,y2)间的欧氏距离：

$$d_{12} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

(2)三维空间两点 a(x1,y1,z1)与 b(x2,y2,z2)间的欧氏距离：

$$d_{12} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

(3)两个 n 维向量 a(x11,x12,…,x1n)与 b(x21,x22,…,x2n)间的欧氏距离：

$$d_{12} = \sqrt{\sum_{k=1}^{n} (x_{1k} - x_{2k})^2}$$

表示成向量运算的形式：

$$d_{12} = \sqrt{(a - b)(a - b)^T}$$

(4)欧式距离公式的 python 实现：

```
# -*- coding: utf-8 -*-
# Filename : vect_distance.py

from numpy import *
import numpy.linalg
import operator
import matplotlib.pyplot as plot
import scipy.spatial.distance as dist

# 向量
```

```
vect1 = mat([1,2,3]); vect2 = mat([6,5,4])

#----各种距离公式----#
#  欧式距离: distO = sqrt( (x1-x2)^2+(y1-y2)^2+(z1-z2)^2 )
pw2V = power(vect1-vect2,2)
distO = sqrt(sum(pw2V))
print "disO:",distO
print
```

输出：

```
disO: 5.9160797831
```

2. 曼哈顿距离(Manhattan Distance)

从名字就可以猜出这种距离的计算方法了。想象你在曼哈顿要从一个十字路口开车到另外一个十字路口，驾驶距离是两点间的直线距离吗？显然不是，除非你能穿越大楼。实际驾驶距离就是这个“曼哈顿距离”。而这也是曼哈顿距离名称的来源， 曼哈顿距离也称为**城市街区距离(City Block distance)**。

(1)二维平面两点 a(x1,y1)与 b(x2,y2)间的曼哈顿距离

$$d_{12} = |x_1 - x_2| + |y_1 - y_2|$$

(2)两个 n 维向量 a(x11,x12,…,x1n)与 b(x21,x22,…,x2n)间的曼哈顿距离

$$d_{12} = \sum_{k=1}^{n} |x_{1k} - x_{2k}|$$

(3)曼哈顿距离 python 实现：

```
#  曼哈顿距离(Manhattan Distance): d(i,j)=|xi-xj|+|yi-yj|
b1 = mat([0,0]);b2=mat([1,0]);b3=mat([0,2])
disM = [sum(abs(b1-b2)),sum(abs(b1-b3)),sum(abs(b3-b2))]
print "disM:",disM
print
```

输出：

```
disM: [1, 2, 3]
```

3. 切比雪夫距离(Chebyshev Distance)

国际象棋玩过么？国王走一步能够移动到相邻的 8 个方格中的任意一个。那么国王从格子(x_1,y_1)走到格子(x_2,y_2)最少需要多少步？自己走走试试。你会发现最少步数总是 max(|x_2-x_1| , |y_2-y_1|) 步。有一种类似的一种距离度量方法叫切比雪夫距离。

(1)二维平面两点 a(x_1,y_1)与 b(x_2,y_2)间的切比雪夫距离：

$$d_{12} = \max(|x_1 - x_2|, |y_1 - y_2|)$$

(2)两个 n 维向量 a(x_{11},x_{12},...,x_{1n})与 b(x_{21},x_{22},...,x_{2n})间的切比雪夫距离：

$$d_{12} = \max_i(|x_{1i} - x_{2i}|)$$

这个公式的另一种等价形式是：

$$d_{12} = \lim_{k\to\infty}\left(\sum_{i=1}^{n}|x_{1i} - x_{2i}|^k\right)^{1/k}$$

看不出两个公式是等价的？提示一下：试试用放缩法和夹逼法则来证明。

(3)切比雪夫距离 Python

```
# 切比雪夫距离
b1 = mat([0,0]);b2=mat([1,0]);b3=mat([0,2])
disCh = [abs(b1-b2).max(),abs(b1-b3).max(),abs(b3-b2).max()]
print "disCh:",disCh
print
```

输出：

```
disCh: [1, 2, 2]
```

4. 闵可夫斯基距离(Minkowski Distance)

闵氏距离不是一种距离，而是一组距离的定义。

(1) 闵氏距离的定义:

两个 n 维变量 a(x_{11},x_{12},...,x_{1n})与 b(x_{21},x_{22},...,x_{2n})间的闵可夫斯基距离定义为：

$$d_{12} = \sqrt[p]{\sum_{k=1}^{n}|x_{1k} - x_{2k}|^p}$$

其中 p 是一个变参数。

当 p=1 时，就是曼哈顿距离

当 p=2 时，就是欧氏距离

当 p→∞时，就是切比雪夫距离

根据变参数的不同，闵氏距离可以表示一类的距离。

(2)闵氏距离的缺点

闵氏距离，包括曼哈顿距离、欧氏距离和切比雪夫距离都存在明显的缺点。

举个例子：二维样本(身高,体重)，其中身高范围是 150~190，体重范围是 50~60，有三个样本：a(180,50)，b(190,50)，c(180,60)。那么 a 与 b 之间的闵氏距离（无论是曼哈顿距离、欧氏距离或切比雪夫距离）等于 a 与 c 之间的闵氏距离，但是身高的 10cm 真的等价于体重的 10kg 么？因此用闵氏距离来衡量这些样本间的相似度很有问题。

简单说来，闵氏距离的缺点主要有两个：

(1)将各个分量的量纲(scale)，也就是"单位"当作相同的看待了。

(2)没有考虑各个分量的分布（期望，方差等)可能是不同的。

(3)闵氏距离 Python 实现(略)

5. 标准化欧氏距离(Standardized Euclidean distance)

(1)标准欧氏距离的定义

标准化欧氏距离是针对简单欧氏距离的缺点而作的一种改进方案。标准欧氏距离的思路：既然数据各维分量的分布不一样，好吧！那我先将各个分量都"标准化"到均值、方差相等吧。均值和方差标准化到多少呢？这里先复习点统计学知识吧，假设样本集 X 的均值(mean)为 m，标准差(standard deviation)为 s，那么 X 的"标准化变量"表示为：

而且标准化变量的数学期望为 0，方差为 1。因此样本集的标准化过程(standardization)用公式描述就是：

$$X^* = \frac{X - m}{s}$$

标准化后的值 = (标准化前的值 － 分量的均值) /分量的标准差

经过简单的推导就可以得到两个 n 维向量 a(x11,x12,…,x1n)与 b(x21,x22,…,x2n)间的标准化欧氏距离的公式：

$$d_{12} = \sqrt{\sum_{k=1}^{n} \left(\frac{x_{1k} - x_{2k}}{s_k} \right)^2}$$

如果将方差的倒数看成是一个权重，这个公式可以看成是一种**加权欧氏距离(Weighted Euclidean distance)**。

(2)标准化欧氏距离 python 实现

```
# 标准化欧式距离:
# 统计向量的标准化
```

```
meanV1 = mean(vect1);
stdV1 = std(vect1);
m1 = ones((1,len(vect1)))*meanV1
if stdV1!=0:
    newVect1 = (vect1-m1)/stdV1
else: newVect1 = (vect1-m1)
meanV2 = mean(vect2);
stdV2 = std(vect2);
m2 = ones((1,len(vect2)))*meanV2
if stdV2!=0:
    newVect2 = (vect2-m2)/stdV2
else: newVect2 = (vect2-m2)

pw2V = power(newVect1-newVect2,2)
stddistO = sqrt(sum(pw2V))
print "stddisO:",stddistO
print
```

输出：

```
stddisO: 3.46410161514
```

6. 马氏距离(Mahalanobis Distance)

（1）马氏距离定义

有 M 个样本向量 X_1~X_m，协方差矩阵记为 S，均值记为向量 μ，则其中样本向量 X 到 u 的马氏距离表示为：

$$D(X) = \sqrt{(X-\mu)^T S^{-1}(X-\mu)}$$

而其中向量 X_i 与 X_j 之间的马氏距离定义为：

$$D(X_i,X_j) = \sqrt{(X_i-X_j)^T S^{-1}(X_i-X_j)}$$

若协方差矩阵是单位矩阵（各个样本向量之间独立同分布）,则公式就成了：

$$D(X_i,X_j) = \sqrt{(X_i-X_j)^T (X_i-X_j)}$$

也就是欧氏距离了。

若协方差矩阵是对角矩阵，公式变成了标准化欧氏距离。

(2)马氏距离的优缺点：量纲无关，排除变量之间的相关性的干扰。

(3)马氏距离的 Python 计算

```
# 马氏距离
X = mat([[1,2],[1,3],[2,2],[3,1]])
Y = dist.pdist(X,'mahalanobis',VI=None)
print "dist.mahalanobis:",Y
```

输出：

```
dist.mahalanobis: [ 2.34520788  2.          2.34520788  1.22474487  2.44948974  1.22474487]
```

7. 夹角余弦(Cosine)

几何中夹角余弦可用来衡量两个向量方向的差异，机器学习中借用这一概念来衡量样本向量之间的差异。

(1)在二维空间中向量 A(x1,y1)与向量 B(x2,y2)的夹角余弦公式：

$$\cos\theta = \frac{x_1 x_2 + y_1 y_2}{\sqrt{x_1^2 + y_1^2}\sqrt{x_2^2 + y_2^2}}$$

(2) 两个 n 维样本点 a(x11,x12,…,x1n)和 b(x21,x22,…,x2n)的夹角余弦

类似的，对于两个 n 维样本点 a(x11,x12,…,x1n)和 b(x21,x22,…,x2n)，可以使用类似于夹角余弦的概念来衡量它们间的相似程度。

$$\cos(\theta) = \frac{a \cdot b}{|a|\,|b|}$$

即：

$$\cos(\theta) = \frac{\sum_{k=1}^{n} x_{1k}\, x_{2k}}{\sqrt{\sum_{k=1}^{n} x_{1k}^2}\sqrt{\sum_{k=1}^{n} x_{2k}^2}}$$

夹角余弦取值范围为[-1,1]。夹角余弦越大表示两个向量的夹角越小，夹角余弦越小表示两向量的夹角越大。当两个向量的方向重合时夹角余弦取最大值 1，当两个向量的方向完全相反夹角余弦取最小值-1。

夹角余弦的具体应用可以参阅参考文献[1]。

(3)python 夹角余弦

```
#夹角余弦(Cosine)公式: 计算空间内两点之间的夹角余弦
b1 = mat([1,0]) ;b2 = mat([1,1.732]) ; b3 = mat([-1,0])
cosb12 = sum(b1*b2.T)/(sqrt(sum(power(b1,2)))*sqrt(sum(power(b2,2))))
cosb13 = sum(b1*b3.T)/(sqrt(sum(power(b1,2)))*sqrt(sum(power(b3,2))))
cosb23 = sum(b2*b3.T)/(sqrt(sum(power(b2,2)))*sqrt(sum(power(b3,2))))
print "cosb12:",cosb12,"    cosb13:",cosb13,"      cosb23:",cosb23
print
```

输出：

```
cosb12: 0.500011000363    cosb13: -1.0    cosb23: -0.500011000363
```

8. 汉明距离(Hamming distance)

(1)汉明距离的定义

两个等长字符串 s1 与 s2 之间的汉明距离定义为将其中一个变为另外一个所需要作的最小替换次数。例如字符串"1111"与"1001"之间的汉明距离为 2。

应用：信息编码（为了增强容错性，应使得编码间的最小汉明距离尽可能大）。

(2)汉明距离 python

```
# nonzero
#  海明距离:比较两个串的相似度
# str1 = [1,1,1,0,0];str2 = [0,1,0,1,0]
str1 = mat([1,1,1,0,0]);str2 = mat([0,1,0,1,0])
disH = 0;
#  标准的方法
# for i in range(len(str1)):
#      if str1[i]!=str2[i]: disH += 1
# numpy  提供的快速计算
#  返回相同字串的索引向量
smstr = nonzero((str1-str2)!=0)[0];
disH = shape(smstr)[1]
print "disH:",disH
print
```

输出：

```
disH: 3
```

9. 杰卡德相似系数(Jaccard similarity coefficient)

(1) 杰卡德相似系数

两个集合 A 和 B 的交集元素在 A，B 的并集中所占的比例，称为两个集合的杰卡德相似系数，用符号 J(A,B)表示。

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

杰卡德相似系数是衡量两个集合的相似度一种指标。

(2) 杰卡德距离

与杰卡德相似系数相反的概念是**杰卡德距离(**Jaccard distance)。杰卡德距离可用如下公式表示：

$$J_\delta(A,B) = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

杰卡德距离用两个集合中不同元素占所有元素的比例来衡量两个集合的区分度。

(3) 杰卡德相似系数与杰卡德距离的应用

可将杰卡德相似系数用在衡量样本的相似度上。

样本 A 与样本 B 是两个 n 维向量，而且所有维度的取值都是 0 或 1。例如：A(0111)和 B(1011)。我们将样本看成是一个集合，1 表示集合包含该元素，0 表示集合不包含该元素。

p ： 样本 A 与 B 都是 1 的维度的个数

q ： 样本 A 是 1，样本 B 是 0 的维度的个数

r ： 样本 A 是 0，样本 B 是 1 的维度的个数

s ： 样本 A 与 B 都是 0 的维度的个数

那么样本 A 与 B 的杰卡德相似系数可以表示为：

这里 p+q+r 可理解为 A 与 B 的并集的元素个数，而 p 是 A 与 B 的交集的元素个数。

而样本 A 与 B 的杰卡德距离表示为：

$$J = \frac{p}{p+q+r}$$

(4)杰卡德距离的 Python 实现

```
# 杰卡德距离
X = mat([[1, 1, 0],[1,-1, 0],[-1, 1, 0]])
Y = dist.pdist(X,'jaccard')
print "dist.jaccard:",Y
```

输出：

```
dist.jaccard: [ 0.5    0.5    1. ]
```

10. 相关系数(Correlation coefficient)与相关距离(Correlation distance)

(1) 相关系数的定义

$$\rho_{XY} = \frac{Cov(X,Y)}{\sqrt{D(X)}\sqrt{D(Y)}} = \frac{E((X-EX)(Y-EY))}{\sqrt{D(X)}\sqrt{D(Y)}}$$

相关系数是衡量随机变量 X 与 Y 相关程度的一种方法，相关系数的取值范围是[-1,1]。相关系数的绝对值越大，则表明 X 与 Y 相关度越高。当 X 与 Y 线性相关时，相关系数取值为 1（正线性相关）或-1（负线性相关）。

(2)相关距离的定义：$D_{xy} = 1 - \rho_{XY}$

(3)相关系数的 Python 实现：

```
# 相关系数：手动计算
vect1 = mat([1, 2 ,3 ,4 ]); vect2 = mat([3 ,8 ,7 ,6])
# 相关系数(Correlation coefficient):衡量 X 与 Y 线性相关程度,其绝对值越大,则表明 X 与 Y 相关度越高。
# 相关系数取值为 1（正线性相关）或-1（负线性相关）
mV1 = mean(vect1)
mV2 = mean(vect2)
length = shape(vect1)[1]
# 协方差的分子部分
covV=(vect1-mV1)*(vect2-mV2).transpose()
# 方差的分子部分
dV1 = sum(power(vect1-mV1,2))
dV2 = sum(power(vect2-mV2,2))
# 相关系数就是两个向量的协方差的乘积/方差的乘积
corref = covV/sqrt(dV1*dV2)
print "corref:",corref
# 使用函数计算相关系数
corrcoefV = corrcoef(vect1,vect2)
print "corrcoef(vect1,vect2):",corrcoefV[0,1]
```

输出：

```
corref: [[ 0.47809144]]
corrcoef(vect1,vect2): 0.478091443734
```

11. 信息熵(Information Entropy)

信息熵是衡量分布的混乱程度或分散程度的一种度量。分布越分散(或者说分布越平均)，信息熵就越大。分布越有序（或者说分布越集中），信息熵就越小。

计算给定的样本集 X 的信息熵的公式：

$$\text{Entropy(X)} = \sum_{i=1}^{n} -p_i \log_2 p_i$$

(1) 参数的含义：

n：样本集 X 的分类数

pi：X 中第 i 类元素出现的概率

信息熵越大表明样本集 S 分类越分散，信息熵越小则表明样本集 X 分类越集中。。当 S 中 n 个分类出现的概率一样大时（都是 1/n），信息熵取最大值 $\log_2(n)$。当 X 只有一个分类时，信息熵取最小值 0

(2) 信息熵的 python 实现

```
# 信息熵
classLabel = mat([0,1,1,1,0,1,1,1,0,1,0,0,1,0])
classlength = shape(classLabel)[1]
label0 = float(shape(nonzero((classLabel)==0)[0])[1])
label1 = float(shape(nonzero((classLabel)==1)[0])[1])
pl0 = label0/classlength
pl1 = label1/classlength
shannonEnt0 = -pl0*math.log(pl0,2)
shannonEnt1 = -pl1*math.log(pl1,2)
print "shannonEnt0:",shannonEnt0,"      shannonEnt1:",shannonEnt1
print
```

输出：

```
shannonEnt0: 0.523882466287       shannonEnt1: 0.461345669747
```