实验三 参数估计&非参数估计

- 姓名：马永田
- 学号：2012911
- 专业：计算机科学与技术专业

实验要求

截止日期：11月4日实验课之前

- 以.ipynb形式的文件提交，输出运行结果，并确保自己的代码能够正确运行
- 发送到邮箱：2120220594@mail.nankai.edu.cn

基本要求

生成两个各包含 N=1200 个二维随机向量的数据集合 X1 和 X2 ， 数据集合中随机向量来自于三个分布模型，分别满足均值向量 μ1=[1,4] , μ2=[4,1] , μ3=[8,4] 和协方差矩阵 D1=D2=D3=2**I** ， 其中**I**是2*2的单位矩阵。在生成数据集合 X1 时，假设来自三个分布模型的先验概率相同 ； 而在生成数据集合 X2 时，先验概率如下： p(w1)=0.6 , p(w2)=0.1 , p(w3)=0.3

1. 在两个数据集合上分别应用“似然率测试规则” 、“最大后验概率规则” 进行分类实验，计算分类错误率，分析实验结果。
2. 在两个数据集合上分别应用 h=1 时的方窗核函数或高斯核函数估计方法，应用“似然率测试规则”进行分类实验，计算分类错误率，分析实验结果。

中级要求

1. 根据初级要求中使用的一个核函数，在数据集 X2 上应用交叉验证法，在 h∈[0.1,0.5,1,1.5,2] 中寻找最优的h值。

高级要求

1. 任选一个数据集，在该数据集上应用k-近邻概率密度估计，任选3个k值输出概率密度分布图。

实验流程

基本要求

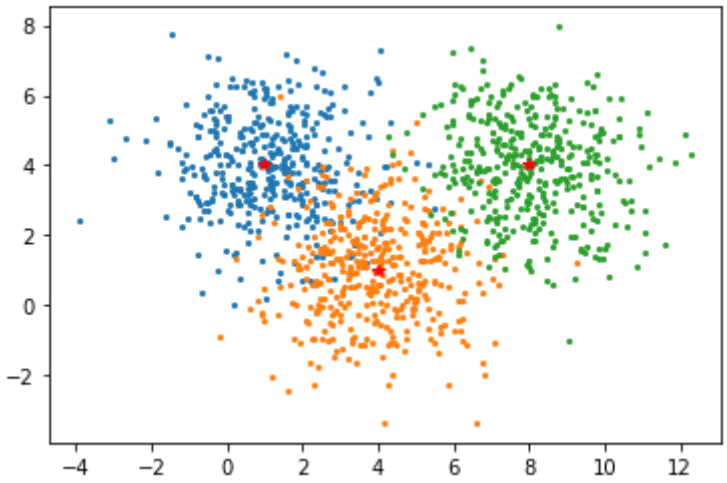
1. 在两个数据集合上分别应用“似然率测试规则” 、“最大后验概率规则” 进行分类实验，计算分类错误率，分析实验结果。
2. 在两个数据集合上分别应用 h=1 时的方窗核函数或高斯核函数估计方法，应用“似然率测试规则”进行分类实验，计算分类错误率，分析实验结果。 #### 生成数据集

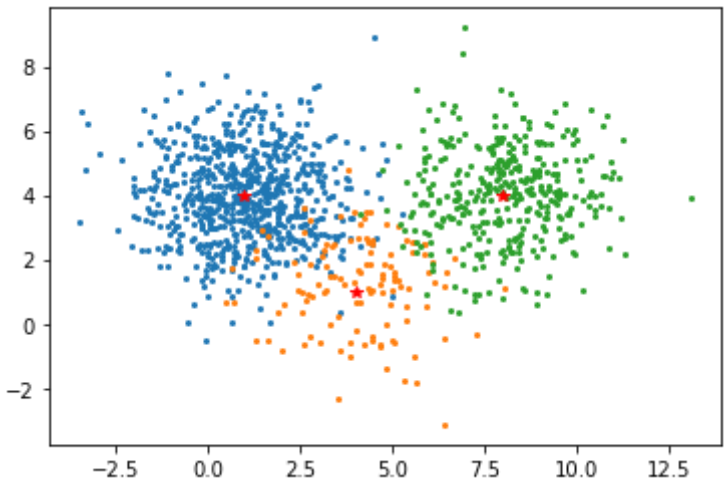
In [1]:
```
# 导入要用到的包
import numpy as np
import sys
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

In [2]:
```
# 生成正态分布数据
def Generate_Sample_Gaussian(mean, cov, P, label):
    '''
        mean 为均值向量
        cov 为方差矩阵a
        P 为单个类的先验概率
        return 单个类的数据集
    '''
    # round(x[,n=0]) 保留到几位小数
    temp_num = round(1200 * P)

    # 生成一个多元正态分布矩阵
    x, y = np.random.multivariate_normal(mean, cov, temp_num).T

    #x,y坐标，x和y矩阵均符合正态分布
    #z表示每个点属于哪一类
    z = np.ones(temp_num) * label
    X = np.array([x, y, z])

     #X.T中每个元素都是有三个元素的列表，分别表示x值，y值，以及对应的标签
    return X.T
```

In [3]:
```
def Generate_DataSet_plot(mean, cov, P):
    # 画出不同先验对应的散点图
    xx = []
    label = 1
    # 将xx变为包含三类数据的数据集
    for i in range(3):
        xx.append(Generate_Sample_Gaussian(mean[i], cov, P[i], label))
        label += 1
        i = i + 1
    #在这时xx是一个有三个元素的列表，每个元素都是一个类
    # 画图
    plt.figure()


    for i in range(3):
        #画出每类的样本向量(x,y)
        plt.plot(xx[i][:, 0], xx[i][:, 1], '.', markersize=4.)
        #画出每类的中心点(均值向量对应的点)
        plt.plot(mean[i][0], mean[i][1], 'r*')
    plt.show()
    return xx
```

In [4]:
```
mean = np.array([[1, 4], [4, 1], [8, 4]])  # 均值数组
cov = [[2, 0], [0, 2]]  # 方差矩阵
num = 1200  # 样本个数
P1 = [1 / 3, 1 / 3, 1 / 3]  # 样本X1的先验概率
P2 = [0.6, 0.1, 0.3]  # 样本X2的先验概率
X1 = np.array(Generate_DataSet_plot(mean, cov, P1), dtype=object)
X2 = np.array(Generate_DataSet_plot(mean, cov, P2), dtype=object)
X1 = np.vstack(X1)
X2 = np.vstack(X2)
X1.shape, X2.shape  # 前两列是坐标，最后一列是标签
```





Out[4]: ((1200, 3), (1200, 3))

参数估计

在两个数据集合上分别应用“似然率测试规则”、“最大后验概率规则”进行分类实验，计算分类错误率，分析实验结果。

In [5]:
```
def Gaussian_function(x, mean, cov):
    det_cov = np.linalg.det(cov)  # 计算方差矩阵的行列式
    inv_cov = np.linalg.inv(cov)  # 计算方差矩阵的逆
    #计算概率p(x|w)
    p = 1/(2*np.pi*np.sqrt(det_cov))*np.exp(-0.5 *
    np.dot(np.dot((x - mean),inv_cov), (x - mean)))
    return p
```

In [6]:
```
# 似然率测试规则
def Likelihood_Test_Rule(X, mean, cov):
    class_num = mean.shape[0]  # 类的个数
    num = np.array(X).shape[0]
    error_rate = 0
    for i in range(num):
        p_temp = np.zeros(3)
        for j in range(class_num):
                # 计算样本i决策到j类的概率
            p_temp[j] = Gaussian_function(X[i][0:2], mean[j], cov)
        p_class = np.argmax(p_temp) + 1  # 得到样本i决策到的类
        if p_class != X[i][2]:
            error_rate += 1
    return round(error_rate / num , 3)
```

In [7]:
```
##最大后验概率规则
def Max_Posterior_Rule(X, mean, cov, P):
    class_num = mean.shape[0]  # 类的个数
    num = np.array(X).shape[0]
    error_rate = 0
    for i in range(num):
        p_temp = np.zeros(3)
        for j in range(class_num):
                # 计算样本i是j类的后验概率
            p_temp[j] =Gaussian_function(X[i][0:2],mean[j],cov)*P[j]
        p_class = np.argmax(p_temp) + 1  # 得到样本i分到的类
        if p_class != X[i][2]:
            error_rate += 1
    return round(error_rate / num,3)
```

In [8]:
```
# 单次试验求不同准则下的分类误差
def repeated_trials(mean, cov, P1, P2, X1, X2):
    # 根据mean, cov, P1,P2生成数据集X1,X2
    # 通过不同规则得到不同分类错误率并返回
    error = np.zeros((2, 2))
    # 计算似然率测试规则误差
    error_likelihood = Likelihood_Test_Rule(X1, mean, cov)
    error_likelihood_2 = Likelihood_Test_Rule(X2, mean, cov)
    error[0] = [error_likelihood, error_likelihood_2]
    # 计算最大后验概率规则误差
    error_Max_Posterior_Rule = Max_Posterior_Rule(X1, mean, cov, P1)
    error_Max_Posterior_Rule_2 = Max_Posterior_Rule(X2, mean, cov, P2)
    error[1] = [error_Max_Posterior_Rule, error_Max_Posterior_Rule_2]
    return error
```

In [9]:
```
#计算误差
error = repeated_trials(mean, cov, P1, P2, X1, X2)
print("data\tLikelihood_Test_Rule\tMax_Posterior_Rule")
print("X1\t", error[0][0], "\t\t\t", error[1][0])
print("X2\t", error[0][1], "\t\t\t", error[1][1])
```

```
data    Likelihood_Test_Rule    Max_Posterior_Rule
X1       0.072                   0.072
X2       0.067                   0.047
```

实验结果分析：

- 不同类别的先验概率P相同时(即数据集X1)，似然率测试规则和最大后验概率规则的分类结果相同
- 当先验概率相差较大时(即数据集X2)，最大后验测试规则更好一些

非参数估计

在两个数据集合上分别应用 h=1 时的方窗核函数或高斯核函数估计方法，应用"似然率测试规则"进行分类实验，计算分类错误率，分析实验结果。

In [10]:
```
# Hint for 初级要求：高斯核概率密度函数计算
# 在公式中，x和mean应该是列向量，但是为了方便，这里接收的都是行向量（维度：1*2）
def Gaussian_Kernel(x, X, h=2):
    # 计算概率p(x|w)
    #print(np.array(x).shape,np.array(X).shape)
    p = (1 / (np.sqrt(2 * np.pi) * h)) * np.array([np.exp(-0.5 * np.dot(x - X[i][0
    return p
```

In [11]:
```
# 高斯核函数 似然率测试规则
```

```
def LikelihoodTest_GaussKernel(X, h=1.0):
    error_rate = 0
    num = X.shape[0]
    w = [[]for i in range(3)]
    for item in X:
        if item[2] == 1:
            w[0].append(item)
        elif item[2] == 2:
            w[1].append(item)
        elif item[2] == 3:
            w[2].append(item)
    for i in range(num):
        p_temp = np.zeros(3)
        for j in range(3):
            p_temp[j] = Gaussian_Kernel(X[i][0:2], w[j], h)
        p_class = np.argmax(p_temp) + 1
        if p_class != X[i][2]:
            error_rate += 1
    return round(error_rate / num, 3)
```

In [12]:
```
error_kernel = np.zeros((2))
error_kernel[0] = LikelihoodTest_GaussKernel(X1)
error_kernel[1] = LikelihoodTest_GaussKernel(X2)
print("Data\t X1\t X2")
print("Error\", error_kernel[0], "\t", error_kernel[1])
```

```
Data     X1      X2
Error    0.069   0.071
```

实验结果分析： 使用高斯核函数估计方法，应用似然率测试规则进行非参数估计，实验中进行多次估计，发现其结果不同于参数估计，在X1与X2两个数据集上的估计结果不存在明显的优劣差别。分析原因是参数估计不假定数学模型，可避免对总体分布的假定不当导致重大错误，所以常有较好的稳健性。

中级要求

1. 根据初级要求中使用的一个核函数，在数据集X2上应用交叉验证法，在 h∈[0.1,0.5,1,1.5,2] 中寻找最优的h值。

In [13]:
```
# 留一法交叉验证
def cross_validate_one(X, h=1.0):
    error_rate = 0
    num = X.shape[0]
    for i in range(num):
        w = [[]for i in range(3)]
        X_train = np.delete(X, i, axis=0)
        for item in X_train:
            if item[2] == 1:
                w[0].append(item)
            elif item[2] == 2:
                w[1].append(item)
            elif item[2] == 3:
                w[2].append(item)
        p_temp = np.zeros(3)
        for j in range(3):
            p_temp[j] = Gaussian_Kernel(X[i][0:2], w[j], h)
        p_class = np.argmax(p_temp) + 1
        if p_class != X[i][2]:
            error_rate += 1
    return round(error_rate / num, 3)
```

In [14]:
```
# 留一法交叉验证
# 觉得上面那个太慢 1200*1200次循环
# 重写了一个看似更快的
# 结果依旧很慢T^T
def cross_validate_one(X, h=1.0):
    error_rate = 0
    num = X.shape[0]
    w = [[]for i in range(3)]
    for item in X:
        if item[2] == 1:
            w[0].append(item)
        elif item[2] == 2:
            w[1].append(item)
        elif item[2] == 3:
            w[2].append(item)
    w = np.array(w,dtype=object)
    w_temp = np.copy(w)
    for deleteW in range(3):
        for i in range(len(w_temp[deleteW])):
            w_temp[deleteW] = np.delete(w_temp[deleteW], i, axis=0)
            p_temp = np.zeros(3)
            for j in range(3):
                p_temp[j] = Gaussian_Kernel(w[deleteW][i][0:2], w_temp[j], h)
            p_class = np.argmax(p_temp) + 1
            if p_class != w[deleteW][i][2]:
                error_rate += 1
            w_temp[deleteW] = np.copy(w[deleteW])
    return round(error_rate / num, 3)
```

In [15]:
```
def fit_bandwidth(X, bandwidth):
    err = []
    for bw in bandwidth:
        err.append(cross_validate_one(X, bw))
    err = np.array(err)
    return err
```

In [16]:
```
bandwidth = [0.1, 0.5, 1, 1.5, 2]
err = fit_bandwidth(X2, bandwidth)
for i in range(5):
    print("h = ",format(bandwidth[i], '.2f'),"\terr = ",err[i])
```

```
h =  0.10       err =  0.079
h =  0.50       err =  0.07
h =  1.00       err =  0.072
h =  1.50       err =  0.074
h =  2.00       err =  0.075
```

实验结果分析： 如上述实验结果可见，当h=0.5时分类效果最好。

高级要求

1. 任选一个数据集，在该数据集上应用k-近邻概率密度估计，任选3个k值输出概率密度分布图。 #### 绘制概率密度分布图示例

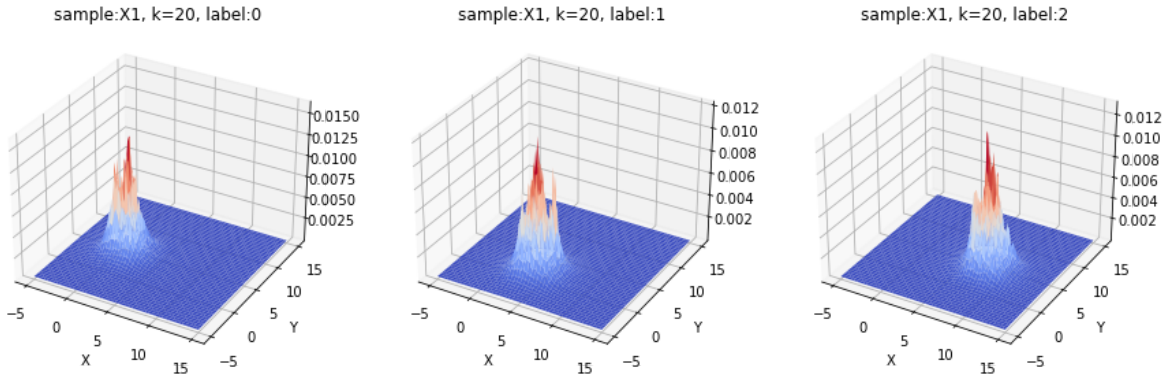
In [17]:
```
def distance(x,y):
    z=np.sqrt((x[0]-y[0])**2+(x[1]-y[1])**2)
    return z
```

In [18]:
```
def Kneibor_Eval(X, k):
    num = len(X)
```

```
    Xtrain = np.array(X)
    w = [[]for i in range(3)]
    for item in Xtrain:
        if item[2] == 1:
            w[0].append(item)
        elif item[2] == 2:
            w[1].append(item)
        elif item[2] == 3:
            w[2].append(item)

    # 生成200*200=40000个采样点，每个采样点对应三类的不同概率
    p = np.zeros((200, 200, 3))
    # 在[-5,15]的范围内，以0.1为步长估计概率密度
    for i in np.arange(0, 200):
        for j in np.arange(0, 200):
            '''
            # 生成标准差距离
            # 根据第k个数据点的位置计算V
            # 找到前k个数据点的类别,分别加到对应类的权重上
            # 计算每个采样点的概率密度函数

            '''
            x=[-5+0.1*i,-5+0.1*j]
            t = k / num
            for w_class in range(3):
                dis = []
                for item in w[w_class]:
                    dis.append(distance(x,item))
                dis.sort()
                v = (dis[k] * np.pi) ** 2
                p[i][j][w_class] = t / v
    return p
```

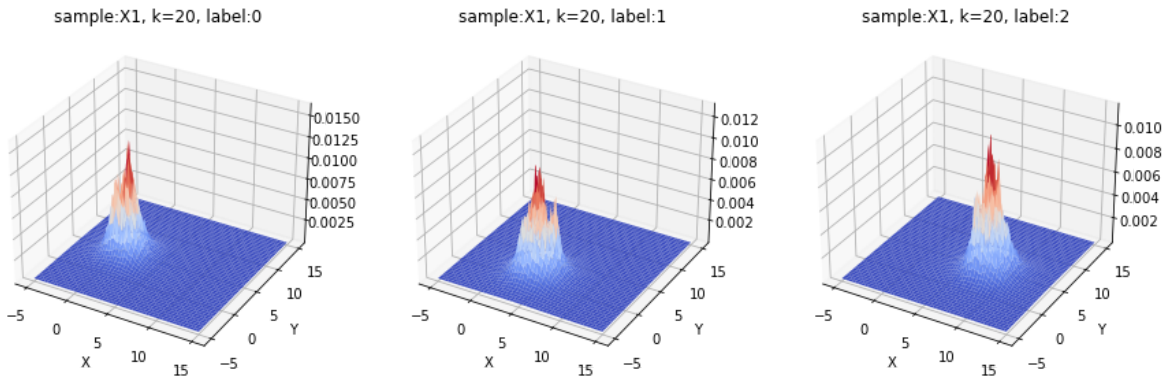
In [19]:
```
p = Kneibor_Eval(X1, 15)  # 获得概率密度估计

X,Y = np.mgrid[-5:15:200j, -5:15:200j]

Z0 = p[:, :, 0]
Z1 = p[:, :, 1]
Z2 = p[:, :, 2]
```

In [20]:
```
fig = plt.figure(figsize=(15,5))
ax = plt.subplot(1, 3, 1,projection='3d')
ax.plot_surface(X, Y, Z0,cmap=plt.cm.coolwarm)
ax.set_title("sample:X1, k=20, label:0")
ax.set_xlabel('X')
ax.set_ylabel('Y')

ax = plt.subplot(1, 3, 2,projection='3d')
ax.plot_surface(X, Y, Z1,cmap=plt.cm.coolwarm)
ax.set_title("sample:X1, k=20, label:1")
ax.set_xlabel('X')
ax.set_ylabel('Y')

ax = plt.subplot(1, 3, 3,projection='3d')
ax.plot_surface(X, Y, Z2,cmap=plt.cm.coolwarm)
ax.set_title("sample:X1, k=20, label:2")
ax.set_xlabel('X')
ax.set_ylabel('Y')
plt.show()
```



In [21]:
```
p = Kneibor_Eval(X1, 20)  # 获得概率密度估计

X,Y = np.mgrid[-5:15:200j, -5:15:200j]

Z0 = p[:, :, 0]
Z1 = p[:, :, 1]
Z2 = p[:, :, 2]
```

In [22]:
```
fig = plt.figure(figsize=(15,5))
ax = plt.subplot(1, 3, 1,projection='3d')
ax.plot_surface(X, Y, Z0,cmap=plt.cm.coolwarm)
ax.set_title("sample:X1, k=20, label:0")
ax.set_xlabel('X')
ax.set_ylabel('Y')

ax = plt.subplot(1, 3, 2,projection='3d')
ax.plot_surface(X, Y, Z1,cmap=plt.cm.coolwarm)
ax.set_title("sample:X1, k=20, label:1")
ax.set_xlabel('X')
ax.set_ylabel('Y')

ax = plt.subplot(1, 3, 3,projection='3d')
ax.plot_surface(X, Y, Z2,cmap=plt.cm.coolwarm)
ax.set_title("sample:X1, k=20, label:2")
ax.set_xlabel('X')
ax.set_ylabel('Y')
plt.show()
```



In [23]:
```
p = Kneibor_Eval(X1, 25)  # 获得概率密度估计

X,Y = np.mgrid[-5:15:200j, -5:15:200j]

Z0 = p[:, :, 0]
Z1 = p[:, :, 1]
Z2 = p[:, :, 2]
```

In [24]:
```
fig = plt.figure(figsize=(15,5))
```

```
ax = plt.subplot(1, 3, 1,projection='3d')
ax.plot_surface(X, Y, Z0,cmap=plt.cm.coolwarm)
ax.set_title("sample:X1, k=20, label:0")
ax.set_xlabel('X')
ax.set_ylabel('Y')

ax = plt.subplot(1, 3, 2,projection='3d')
ax.plot_surface(X, Y, Z1,cmap=plt.cm.coolwarm)
ax.set_title("sample:X1, k=20, label:1")
ax.set_xlabel('X')
ax.set_ylabel('Y')

ax = plt.subplot(1, 3, 3,projection='3d')
ax.plot_surface(X, Y, Z2,cmap=plt.cm.coolwarm)
ax.set_title("sample:X1, k=20, label:2")
ax.set_xlabel('X')
ax.set_ylabel('Y')
plt.show()
```