机器学习基础 实验二 实验报告

2021级软件5班 马贵亮 202122202214

实验目的

了解线性判别函数与参数、非参数估计的相关知识，完成如下四个任务。

实验内容

将在后续实验过程中依次阐述对应的内容。

实验过程

1 线性判别函数

1.1实验内容

采用`exp2_1.mat`中的数据，实现线性判别函数分类算法，其`x1、x2`为二维自变量，`y`为样本类别。编程实现线性判别函数分类，并做出分类结果可视化。

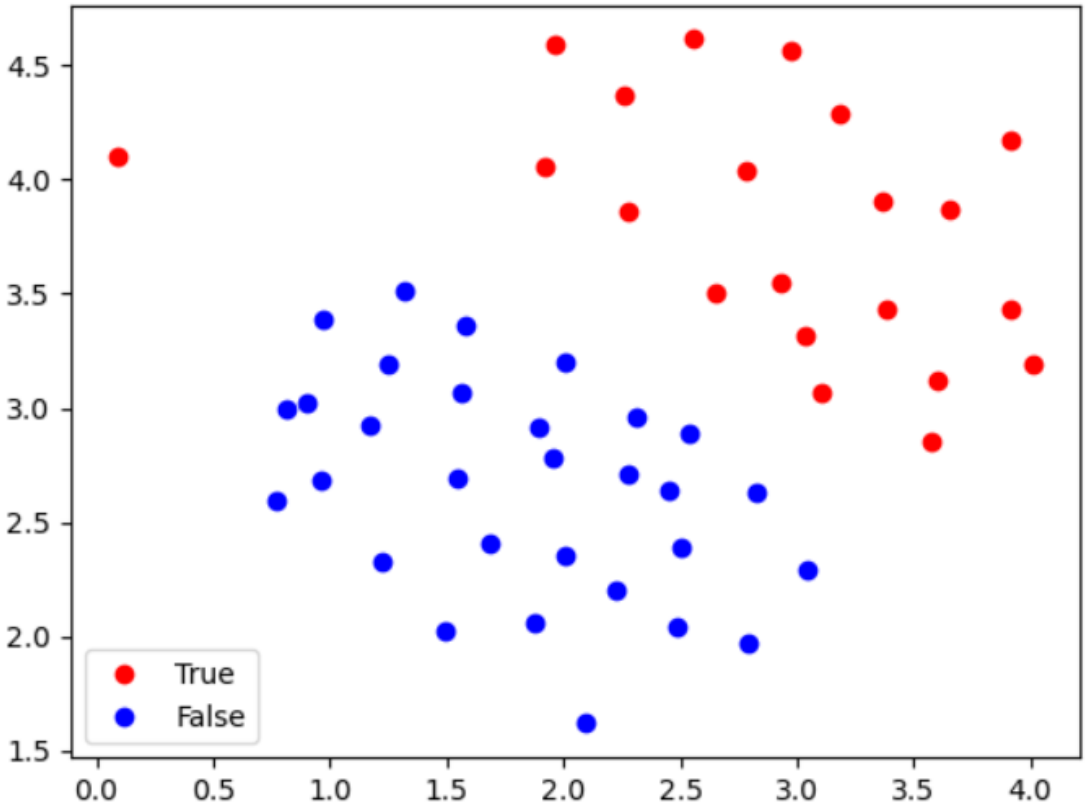
1.2 实验过程

首先我需要对数据进行读取，由于原始数据存在在一个`.mat`文件中，因此需要借助`scipy`库对`.mat`文件进行读入，读入后为了后续处理方便，我在该代码中将其转换为`dataFrame`来便于我后续的数据处理。

有关代码如下：

```
import scipy
import pandas as pd
data = scipy.io.loadmat('../../data/e2data1.mat')
X = pd.DataFrame(data['X'], columns=['x1', 'x2'])
y = pd.DataFrame(data['y'], columns=['y'])
```

我们执行完这段代码后我们便有了一组数据，首先根据`y`对应的值将所有的`X`分成两类，由于所有`X`均为二维散点，因此可以根据不同的颜色绘制散点图，来实现对数据的基本可视化。



通过对散点分布的大致观察，该组数据大致线性可分，因此采用简单的线性分类器来对其进行线性判别。针对线性判别，我设计了一个 `LinearClassifier` 类来实现线性判别模型的构建。

在原理方面，我采用单个感知机的模型进行实现，该感知机的激活函数设计为 `sigmoid` 函数。即：$Sigmoid(x) = \frac{1}{1+e^{-x}}$。损失函数采用交叉熵即：$Loss(z) = ylnz + (1-y)ln(1-z)$。设该线性模型的权重为 ω，偏置为b。则整体模型为 $z = Sigmoid(X\omega + b)$。而最终判别函数以 0.5为界，大于者为正例，小于者为负例。整体损失为$Cost = \frac{1}{n}\sum Loss(z)$

我们的目的是使得损失值尽可能小，由梯度下降则

$$d\omega = \frac{dCoss}{dz} \cdot Sigmoid'(X\omega + b) \cdot X^T = \frac{1}{n}X^T(z-y)$$

则下降方向为$-d\omega = -\frac{1}{n}X^T(z-y)$

设学习率为α 则每次更新为 $\omega = \omega - \alpha d\omega$

再设计一个迭代轮次，则可以实现整个线性分类器的设计，在具体实现的过程中将 X，转换成 $[1 \quad X]$，整体代码如下：

$Sigmoid$ 函数实现，以及数据转换代码：

```python
'''
DataProcessor.py
'''
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


def PrepareForTraining(df):
    df.insert(0,'',1)
    return df
```

```
12
13 def Sigmoid(x):
14     return 1 / (1 + np.exp(-x))
15
```

线性分类设计：

```
1  '''
2  LinearClassifier.py
3  '''
4  import numpy as np
5  from DataProcessor import PrepareForTraining, Sigmoid
6
7
8  class LinearClassifier:
9      def __init__(self, alpha=0.01, iterations=1000):
10         self.alpha = alpha
11         self.iterations = iterations
12         self.weights = None
13
14     def gradientDescent(self, X, y):
15         n = X.shape[0]
16         pred = Sigmoid(np.dot(X, self.weights))
17         delta = pred - y
18         self.weights -= self.alpha * (1 / n) * np.dot(X.T, delta) #交叉熵
   $L(w) = -yln[Sigmoid(Xw)]-(1-y)ln[1-Sigmoid(Xw)]$
19         # self.weights -= self.alpha * (1 / n) * np.dot(X.T, delta * pred *
   (1 - pred)) # $L(w)=\frac{1}{2n}[Sigmoid(Xw)-y]^2$
20
21     def fit(self, X, y):
22         X = PrepareForTraining(X)
23         n, m = X.shape
24         self.weights = np.zeros([m, 1])
25         for _ in range(self.iterations):
26             self.gradientDescent(X, y)
27         return self.weights
28
29     def predict(self, X):
30         X = PrepareForTraining(X)
31         return Sigmoid(np.dot(X, self.weights))>=0.5
```

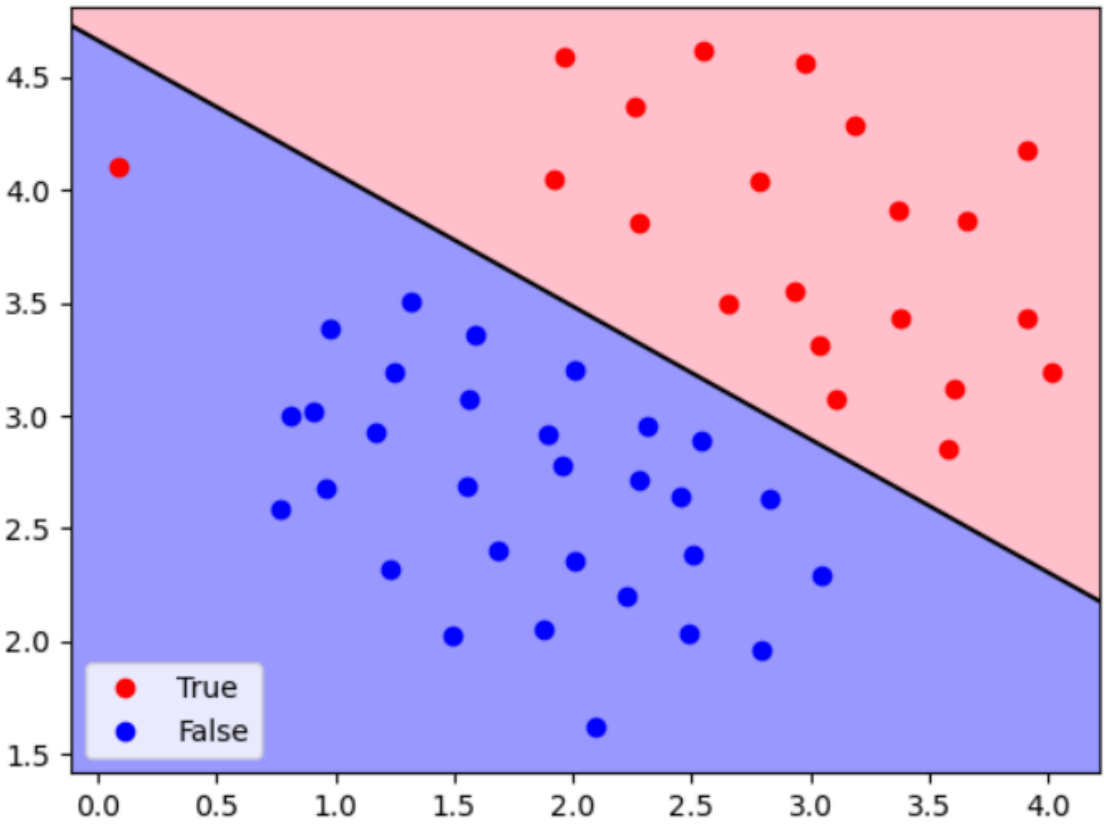
随后我对我自己的代码进行调用和训练，可以获得对应的权重

```
1  from LinearClassifier import LinearClassifier
2  from DataProcessor import PrepareForTraining, LinearBoundary
3
4  LC = LinearClassifier(alpha=0.1, iterations=10000)
5  weight = LC.fit(X, y)
```

最终通过简单的数据处理，在该区间内绘制出线性分类直线，绘制对应的分类染色，基于如下代码实现。最终在执行后可以获得如下图。

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  import pandas as pd
4  from matplotlib.colors import ListedColormap
```

```
5
6  x_min, x_max = X['x1'].min() - 0.2, X['x1'].max() + 0.2
7  y_min, y_max = X['x2'].min() - 0.2, X['x2'].max() + 0.2
8  x = np.linspace(x_min, x_max, 100)
9  liny = LinearBoundary(weight, x)
10 plt.plot(x, liny, 'k')
11
12 custom_cmap = ListedColormap(['#9898ff',  # 浅红
13                               '#FFC0CB',])
14
15 x0, x1 = np.meshgrid(np.linspace(x_min, x_max, 1000).reshape(-1, 1),
16                      np.linspace(y_min, y_max, 1000).reshape(-1, 1))
17 X_new = np.c_[x0.ravel(), x1.ravel()]
18 X_new = pd.DataFrame(X_new)
19 Y_new = LC.predict(X_new)
20 z = Y_new.reshape(x0.shape)
21 plt.contourf(x0, x1, z, cmap=custom_cmap)
22
23 plt.axis([x_min, x_max, y_min, y_max])
24 plt.legend(loc='lower left')
25 plt.show()
```



即可实现对实验该部分的实现。

2 最大似然估计

2.1 实验内容

掌握用最大似然估计进行参数估计的原理；当训练样本服从多元正态分布时，计算不同高斯情况下的均值和方差。

使用上面给出的三维数据或者使用 `exp2_2.xlsx` 中的数据：

(1) 编写程序，对类 1 和类 2 中的三个特征x_i分别求解最大似然估计的均值$\hat{\mu}$和方差σ^2。

(2) 编写程序，处理二维数据的情形$p(x) \sim N(\mu, \sum)$。对类 1 和类 2 中任意两个特征的组合分别求解最大似然估计的均值 $\hat{\mu}$ 和方差 $\hat{\sum}$（每个类有3种可能）。

(3) 编写程序，处理三维数据的情形$p(x) \sim N(\mu, \sum)$。对类 1 和类 2 中三个特征求解最大似然估计的均值 $\hat{\mu}$ 和方差$\hat{\sum}$。

(4) 假设该三维高斯模型是可分离的，即$\sum = diag(\sigma_1^2, \sigma_2^2, \sigma_3^2)$，编写程序估计类 1 和类 2 中的均值和协方差矩阵中的参数。

(5) 比较前 4 种方法计算出来的每一个特征的均值μ_i的异同，并加以解释。

(6) 比较前 4 种方法计算出来的每一个特征的方差σ_i的异同，并加以解释。

2.2 实验过程

首先是基本的数据读入，由于该组数据为 `.xlsx` 文件，因此采用 `pandas` 可以对其进行读入，读入后后再做剩余的处理，采用最简单的 `pd.read_excel` 来实现，并且利用 `y` 对应的值对其进行分类以便后边所有题的处理。

```
df = pd.read_excel('../../data/exp2-2.xlsx')
y = df['y']
X = df.drop('y', axis=1)
X1 = X[y == 1]
X2 = X[y == 0]
```

对于（1），对于每个类别的每个特征 x_i

$$\mu = \frac{1}{n} x_i$$

$$\sigma^2 = \frac{1}{n} (x_i - \mu)^2$$

同样在python的 `numpy` 包中，有着 `mean` 和 `var` 两个函数可以用来计算μ 和 σ^2，对此我设计了如下代码来进行计算和比较

```
def Part1(data):
    '''
        编写程序，对类 1 和类 2 中的三个特征xi分别求解最大似然估计的均值μ̂和方差σ̂2。
        当前要处理的工作即将类中三个特征各自看作独立，求其均值和方差
        即此时 μ和 Σ 均不知
    '''
    # 手动实现计算均值和方差
    data = data.to_numpy()
    n, m = data.shape
    miu = np.zeros(m)
    sigma2 = np.zeros(m)
    for x in data:
        miu += x / n
    for x in data:
```

```
        sigma2 += (x - miu) ** 2 / n
    print(f'当对三个特征分别手算求最大似然估计时:\nmiu=\n{miu}\nsigma2=\n{sigma2}')

    # 也可以采用pandas中的包来计算均值和方差
    miu = np.mean(data, axis=0)
    sigma2 = np.var(data, axis=0, ddof=0)
    print(f'当对三个特征分别利用numpy求最大似然估计时:\nmiu=\n{miu}\nsigma2=\n{sigma2}\n')

print('第一部分:')
print("For Class 1:")
Part1(X1)
print("For Class 2:")
Part1(X2)
```

得到的运行结果如下

```
第一部分:
For Class 1:
当对三个特征分别手算求最大似然估计时:
miu=
[-0.0709 -0.6047 -0.911 ]
sigma2=
[0.90617729 4.20071481 4.541949  ]
当对三个特征分别利用numpy求最大似然估计时:
miu=
[-0.0709 -0.6047 -0.911 ]
sigma2=
[0.90617729 4.20071481 4.541949  ]

For Class 2:
当对三个特征分别手算求最大似然估计时:
miu=
[-0.1216   0.4299   0.00372]
sigma2=
[0.05820804 0.04597009 0.00726551]
当对三个特征分别利用numpy求最大似然估计时:
miu=
[-0.1216   0.4299   0.00372]
sigma2=
[0.05820804 0.04597009 0.00726551]
```

通过比对发现，两者的计算完全一致。

对于 (2) 的部分，我依次枚举两种组合，并以此计算其组合后各自的μ 和 $\sum$。

对于一个两两组合的数据，其 μ 的求法与上述基本一致，只不过换成了矩阵运算。而对于 $\sum = \frac{1}{n}\sum(x_i-\mu)\cdot(x_i-\mu)^T$。并且也尝试采用python中的`mean`方法和`cov`方法来进行求解。代码和输出如下

```
def DealForMatrix(data):
    data = data.to_numpy()
    n, m = data.shape
    miu = np.zeros(m)
    sigma2 = np.zeros([m, m])
    for x in data:
```

```
        miu += x
    miu = miu / n
    for x in data:
        delta = (x - miu).reshape(-1, 1)
        sigma2 += np.dot(delta, delta.T)
    sigma2 = sigma2 / n
    print(f'当手算求最大似然估计时:\nmiu=\n{miu}\nsigma2=\n{sigma2}')

    # 也可以采用pandas中的包来计算均值和方差
    miu = np.mean(data, axis=0)
    sigma2 = np.cov(data, rowvar=False, ddof=0)
    print(f'当利用numpy求最大似然估计时:\nmiu=\n{miu}\nsigma2=\n{sigma2}\n')


def Part2(data):
    n, m = data.shape
    for i in range(m):
        for j in range(i + 1, m):
            print(f'采用x{i + 1}和x{j + 1}进行计算所得:')
            tmp = pd.concat([data.iloc[:, i], data.iloc[:, j]],
axis=1).copy()
            DealForMatrix(tmp)

print("第二部分:")
print("For Class 1:")
Part2(X1)
print("For Class 2:")
Part2(X2)
```

```
第二部分:
For Class 1:
采用x1和x2进行计算所得:
当手算求最大似然估计时:
miu=
[-0.0709 -0.6047]
sigma2=
[[0.90617729 0.56778177]
 [0.56778177 4.20071481]]
当利用numpy求最大似然估计时:
miu=
[-0.0709 -0.6047]
sigma2=
[[0.90617729 0.56778177]
 [0.56778177 4.20071481]]

采用x1和x3进行计算所得:
当手算求最大似然估计时:
miu=
[-0.0709 -0.911 ]
sigma2=
[[0.90617729 0.3940801 ]
 [0.3940801  4.541949  ]]
当利用numpy求最大似然估计时:
miu=
[-0.0709 -0.911 ]
sigma2=
[[0.90617729 0.3940801 ]
```

```
 [0.3940801  4.541949  ]]

采用x2和x3进行计算所得:
当手算求最大似然估计时:
miu=
[-0.6047 -0.911 ]
sigma2=
[[4.20071481 0.7337023 ]
 [0.7337023  4.541949  ]]
当利用numpy求最大似然估计时:
miu=
[-0.6047 -0.911 ]
sigma2=
[[4.20071481 0.7337023 ]
 [0.7337023  4.541949  ]]

For Class 2:
采用x1和x2进行计算所得:
当手算求最大似然估计时:
miu=
[-0.1216  0.4299]
sigma2=
[[ 0.05820804 -0.01321216]
 [-0.01321216  0.04597009]]
当利用numpy求最大似然估计时:
miu=
[-0.1216  0.4299]
sigma2=
[[ 0.05820804 -0.01321216]
 [-0.01321216  0.04597009]]

采用x1和x3进行计算所得:
当手算求最大似然估计时:
miu=
[-0.1216   0.00372]
sigma2=
[[ 0.05820804 -0.00478645]
 [-0.00478645  0.00726551]]
当利用numpy求最大似然估计时:
miu=
[-0.1216   0.00372]
sigma2=
[[ 0.05820804 -0.00478645]
 [-0.00478645  0.00726551]]

采用x2和x3进行计算所得:
当手算求最大似然估计时:
miu=
[0.4299  0.00372]
sigma2=
[[0.04597009 0.00850987]
 [0.00850987 0.00726551]]
当利用numpy求最大似然估计时:
miu=
[0.4299  0.00372]
sigma2=
[[0.04597009 0.00850987]
 [0.00850987 0.00726551]]
```

可以发现其计算结果保持一致，即代码设计正确。

对于 (3) 部分，其关键步骤于 (2) 保持一致，直接调用`DealForMatrix`即可，其具体代码和结果如下：

```
def Part3(data):
    DealForMatrix(data)

print("第三部分:")
print("For Class 1:")
Part3(X1)
print("For Class 2:")
Part3(X2)
```

```
第三部分:
For Class 1:
当手算求最大似然估计时:
miu=
[-0.0709 -0.6047 -0.911 ]
sigma2=
[[0.90617729 0.56778177 0.3940801 ]
 [0.56778177 4.20071481 0.7337023 ]
 [0.3940801  0.7337023  4.541949  ]]
当利用numpy求最大似然估计时:
miu=
[-0.0709 -0.6047 -0.911 ]
sigma2=
[[0.90617729 0.56778177 0.3940801 ]
 [0.56778177 4.20071481 0.7337023 ]
 [0.3940801  0.7337023  4.541949  ]]

For Class 2:
当手算求最大似然估计时:
miu=
[-0.1216   0.4299   0.00372]
sigma2=
[[ 0.05820804 -0.01321216 -0.00478645]
 [-0.01321216  0.04597009  0.00850987]
 [-0.00478645  0.00850987  0.00726551]]
当利用numpy求最大似然估计时:
miu=
[-0.1216   0.4299   0.00372]
sigma2=
[[ 0.05820804 -0.01321216 -0.00478645]
 [-0.01321216  0.04597009  0.00850987]
 [-0.00478645  0.00850987  0.00726551]]
```

对于第四部分，当$\sum = diag(\sigma_1^2, \sigma_2^2, \sigma_3^2)$时，其对应的$\sigma$和第一部分计算保持一致，因此不再过多赘述。

```
def Part4(data):
    '''
    绝大多数代码在Part1中已经呈现，因此此处直接使用numpy实现
    '''
    data = data.to_numpy()
```

```
    miu = np.mean(data,axis=0)
    sigma2 = np.var(data,axis=0,ddof=0)
    cov = np.diag(sigma2)
    print(f'当利用numpy求最大似然估计时:\nmiu=\n{miu}\ncov=\n{cov}\n')

print("第四部分:")
print("For Class 1:")
Part4(X1)

print("For Class 2:")
Part4(X2)
```

```
第四部分:
For Class 1:
当利用numpy求最大似然估计时:
miu=
[-0.0709 -0.6047 -0.911 ]
cov=
[[0.90617729 0.         0.        ]
 [0.         4.20071481 0.        ]
 [0.         0.         4.541949  ]]

For Class 2:
当利用numpy求最大似然估计时:
miu=
[-0.1216   0.4299   0.00372]
cov=
[[0.05820804 0.         0.        ]
 [0.         0.04597009 0.        ]
 [0.         0.         0.00726551]]
```

对于 (5) 和(6)在求解均值和协方差的过程中，计算最大似然估计，协方差计算中除以`n`是合适的。即采用有偏估计。整体的值均保持一致。

- 对比均值：各种方法计算的均值应该一致，因为均值的估计与协方差矩阵的形式无关。
- 对比方差：直接方差估计与在协方差矩阵中提取的方差应一致。然而，对角协方差矩阵只考虑单个变量的方差，忽略变量之间的相关性，可能在某些应用中提供不同的视角。

3-Parzen窗

3-Parzen.1 实验内容

使用上面表格中的数据或者使用 exp2_3.xlsx 中的数据进行 Parzen 窗估计和设计分类器。窗函数为一个球形的高斯函数如公式2-1所示：

$$\varphi(\frac{x-x_i}{h}) \propto exp[-\frac{(x-x_i)^T(x-x_i)}{2h^2}]$$

编写程序，使用 Parzen 窗估计方法对任意一个的测试样本点xx进行分类。对分类器的训练则使用表2-2中的三维数据。令$h=1$，分类样本点为$(0.5,1.0,0.0)^T$，$(0.31,1.51,-0.50)^T$，$(-0.3,0.44,-0.1)^T$。

3-Parzen.2 实验过程

Parzen窗的基本原理为以当前点为中心，绘制一个h的高斯窗口，对其中进行数点，即可返回类条件概率密度，在理想情况下也可根据点数的多少直接进行分类。那么基于如上基础进行了简单代码实现如下：

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


def ParzenWindow(X, h, test):
    X = X.to_numpy()
    sigma = 0
    for x in X:
        sigma += np.exp(-np.dot((test - x), (test - x).T) / 2 * h ** 2)
    return sigma


def Parzen(X, y, h, test):
    test = np.array(test)
    yList = y.drop_duplicates().tolist()
    res = np.zeros([len(yList), 1])
    for index,testX in enumerate(test):
        K = np.zeros(len(yList))
        for i, yEle in enumerate(yList):
            XList = X[y == yEle]
            k = ParzenWindow(XList, h, testX)
            K[i] = k / (XList.shape[0] * h ** XList.shape[1])
        res[index] = yList[np.argmax(K)]
    return res


df = pd.read_excel('../../data/exp2-3.xlsx')
df.to_csv('../../data/exp2-3.csv', index=False)
y = df['y']
X = df.drop('y', axis=1)
Xtest = [[0.5, 1.0, 0.0],
         [0.31, 1.51, -0.50],
         [-0.3, 0.44, -0.1]]
yPred = Parzen(X, y, 1, Xtest)
print(yPred)
```

在进行计算后，可以获得如下结果：

```
[[2.]
 [2.]
 [2.]]
```

则将该过程实现。

3-KNN

3-KNN.1 实验内容

k-近邻概率密度估计：

对上面表格中的数据使用k-近邻方法进行概率密度估计：

1. 编写程序，对于一维的情况，当有n个数据样本点时，进行k-近邻概率密度估计。对表格中的类3的特征x_1，用程序画出当$k=1,3,5$时的概率密度估计结果。
2. 编写程序，对于二维的情况，当有 n 个数据样本点时，进行k-近邻概率密度估计。对表格中的类2的特征$(x_1,x_2)^T$，用程序画出当 $k=1,3,5$时的概率密度估计结果。
3. 编写程序，对表格中的3个类别的三维特征，使用k-近邻概率密度估计方法。并且对下列点处的概率密度进行估计：
   $(-0.41,0.82,0.88)^T$，$(0.14,0.72,4.1)^T$，$(-0.81,0.61,-0.38)^T$。

3-KNN.2 实验过程

对于整个实验我大致有两种不同的理解方案，其实际的预测结果差值不大，但其实际运算的类条件概率密度具有一定的差异性，而且在后段的预测过程中使用了贝叶斯的思想，在预测不仅仅只考虑类条件概率密度，也考虑进先验概率。

我的第一种理解时仅考虑题面所提及的类3（题1），和类2（题2）。仅使用这一个类，利用knn求得这一类得类条件概率密度（与其他类无关）。即k个点只针对一个样本，考虑到如果这个点附近的不够密，那么其距离会很远，则类条件概率密度很小。

使用多个类，但是只展示类3（题1），类2（题2）的概率密度。即k个点是所有样本。那么在这k个点中最多的那类的类条件概率密度应当最高。

再利用求得的类条件概率密度与先验概率相乘再比较大小进行预测。

在计算其中的距离后对应的体积时，我采用了高维球体的体积计算公式进行

超球体体积 $V_n(r)=\frac{\pi^{n/2}}{\Gamma(\frac{n}{2}+1)}r^n$，令 $\alpha=\frac{\pi^{n/2}}{\Gamma(\frac{n}{2}+1)}$为系数，则 $V_n(r)=\alpha r^n$

并且基于我个人对于KNN的理解，构建了一个KNN类来实现该实验以及4 KNN实战的部分的KNN代码，如下：

```
'''
该代码为在实验4内容修改后的knn代码
'''

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from numpy.linalg import inv
import math


class kNNClassifier:

    def KNN_Vn(self, X, k, t):
        '''
        基于 p_{n}(x) = k_{n}/(V_{n}*n) 来计算概率密度
        :param X: 原始数据集的某一类别
        :param k: KNN 的k
        :param t: 测试集
        :return: 一个概率密度函数
```

```
        '''
        res = np.zeros((t.shape[0],))
        for i, test in enumerate(t):
            dist = np.linalg.norm(X - test, axis=1)
            dist = np.sort(dist)
            if dist[k - 1] != 0:
                res[i] = k / (X.shape[0] * dist[k - 1] ** X.shape[1])
            else:
                res[i] = k / (X.shape[0] * (1e-5) ** X.shape[1])
        return res

    def kNN_Vn(self, X_train, y_train, X_test, k):
        res = [{} for _ in range(X_test.shape[0])]
        yValueList = y_train.unique().tolist()
        for y in yValueList:
            trainSet = X_train[y_train == y].copy()
            yRes = self.KNN_Vn(trainSet, k, X_test)
            for i in range(len(yRes)):
                # 为 res 中的每一行的字典添加键值对，键为 y，值为 yRes 中对应的元素
                res[i][y] = yRes[i]
        return res

    def gamma(self, x):
        # 递归利用已知的Gamma(1/2) = sqrt(pi)
        if abs(x - 0.5) < 1e-6:
            return math.sqrt(math.pi)
        elif abs(x - 1) < 1e-6:
            return 1
        else:
            return (x - 1) * self.gamma(x - 1)

    def kNN_Euler_Count(self, X_train, y_train, X_test, k):
        '''
        基于欧拉距离的计数方法来计算类条件概率密度
        :param X_train: 训练集
        :param y_train: 训练标签
        :param X_test: 测试集
        :param k: k近邻的k
        :return: 测试集的类条件概率密度
        '''
        res = []
        # 超球体体积系数 $V_n(r) = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2} + 1)} r^n$ 令 $ alpha = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2} + 1)} $
        pi_power = math.pi ** (X_train.shape[1] / 2)
        gamma_value = self.gamma(((X_train.shape[1] / 2) + 1))
        alpha = ((pi_power) / gamma_value) if gamma_value != 0 else float('inf')
        for i, test in enumerate(X_test):
            dist = np.linalg.norm(X_train - test, axis=1)
            KNN_indices = np.argsort(dist)[:k]
            KNN_labels = y_train[KNN_indices]
            d = dist[KNN_indices[-1]]
            if d == 0:
                d = 1e-5
            class_probs = {
                cls: np.sum(KNN_labels == cls) / (X_train.shape[0] * alpha * (d ** X_train.shape[1]))
                for cls in np.unique(y_train)}
```

```
            res.append(class_probs)
        return res

    def mahalanobis_distance(self, x, dataset, invCov):
        '''
        计算单个样本与数据集所有样本的马氏距离
        :param x:
        :param dataset:
        :param invCov:
        :return:
        '''
        dist = np.zeros(dataset.shape[0])
        dataset = dataset.to_numpy()
        for i, data in enumerate(dataset):
            delta = x - data
            dist[i] = np.sqrt(delta.dot(invCov).dot(delta.T))
        return dist

    def kNN_Mahalanobis_Count(self, X_train, y_train, X_test, k):
        '''
        基于马氏距离的计数方法来计算类条件概率密度
        :param X_train: 训练集
        :param y_train: 训练标签
        :param X_test: 测试集
        :param k: k近邻的k
        :return: 测试集的类条件概率密度
        '''
        # 超球体体积系数 $V_n(r) = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2} + 1)} r^n$ 令 $ alpha = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2} + 1)} $
        pi_power = math.pi ** (X_train.shape[1] / 2)
        gamma_value = self.gamma((X_train.shape[1] / 2) + 1)
        alpha = ((pi_power) / gamma_value) if gamma_value != 0 else float('inf')
        cov = np.cov(X_train.T)
        invCov = inv(cov)
        res = []
        for i, test in enumerate(X_test):
            dist = self.mahalanobis_distance(test, X_train, invCov)
            KNN_indices = np.argsort(dist)[:k]
            KNN_labels = y_train[KNN_indices]
            d = dist[KNN_indices[-1]]
            if d == 0:
                d = 1e-5
            class_probs = {
                cls: np.sum(KNN_labels == cls) / (X_train.shape[0] * alpha * (d ** X_train.shape[1]))
                for cls in np.unique(y_train)}
            res.append(class_probs)
        return res

    def density(self, X_train, y_train, X_test, k, typ=0):
        yValueList = y_train.unique().tolist()
        prior_prob = {}
        if not isinstance(X_test, np.ndarray):
            # 如果不是，转换它为numpy数组
            X_test = np.array(X_test)
        for y in yValueList:
            prior_prob[y] = X_train[y_train == y].shape[0]
```

```
        if typ == 0:
            probs = self.kNN_Euler_Count(X_train, y_train, X_test, k)
        elif typ == 1:
            probs = self.kNN_Mahalanobis_Count(X_train, y_train, X_test, k)
        elif typ == 2:
            probs = self.kNN_Vn(X_train, y_train, X_test, k)
        return probs
```

如上述实验一致，我们需要先对数据进行读入，依旧采用`pandas`包中的`read_excel`方法将其读入为一个`dataFrame`，并将`X`和`y`提取出来，并且导入(3)中需要预测的数据

```
df = pd.read_excel('../../data/exp2-3.xlsx')
df.to_csv('../../data/exp2-3.csv', index=False)
y = df['y']
X = df.drop('y', axis=1)
Xtest = [[-0.41, 0.82, 0.88],
         [0.14, 0.72, 4.1],
         [-0.81, 0.61, -0.38]]
Xtest = np.array(Xtest)
```

以下为两种不同的理解对应的各自结果和图像。

首先对于(1):

对于第一种理解，需要将数据根据类完全摘离，数据处理即为：

```
X1 = X['x1'].copy()
X_train = X1[y == 3].copy().reset_index(drop=True)
y_train = y[y == 3].copy().reset_index(drop=True)
DealPart1(X_train, y_train)
```

对于第二种理解，不需要单独摘出数据，数据预处理为：

```
X1 = X['x1'].copy()
X_train = X1.copy().reset_index(drop=True)
y_train = y.copy().reset_index(drop=True)
DealPart1(X_train, y_train)
```

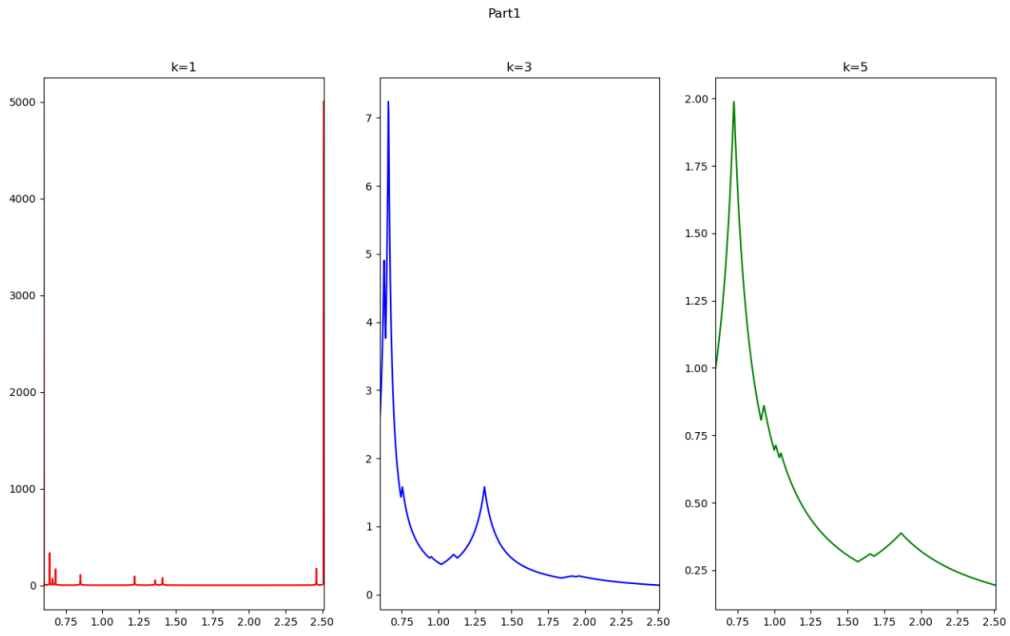
而两者调用KNN的方法保持一致，即如下函数`DealPart1`：

```
def DealPart1(X_train, y_train):
    x_min = X_train.min()
    x_max = X_train.max()
    X_train = X_train.to_numpy().reshape(-1, 1)
    X_test = np.linspace(x_min, x_max, 1000, endpoint=True).reshape(-1, 1)
    knn = kNNClassifier()

    # 获得概率密度函数
    y1_density = knn.density(X_train, y_train, X_test, 1)
    y3_density = knn.density(X_train, y_train, X_test, 3)
    y5_density = knn.density(X_train, y_train, X_test, 5)

    y1_test = np.array([d[3] for d in y1_density]).reshape(-1, 1)
    y3_test = np.array([d[3] for d in y3_density]).reshape(-1, 1)
    y5_test = np.array([d[3] for d in y5_density]).reshape(-1, 1)
```

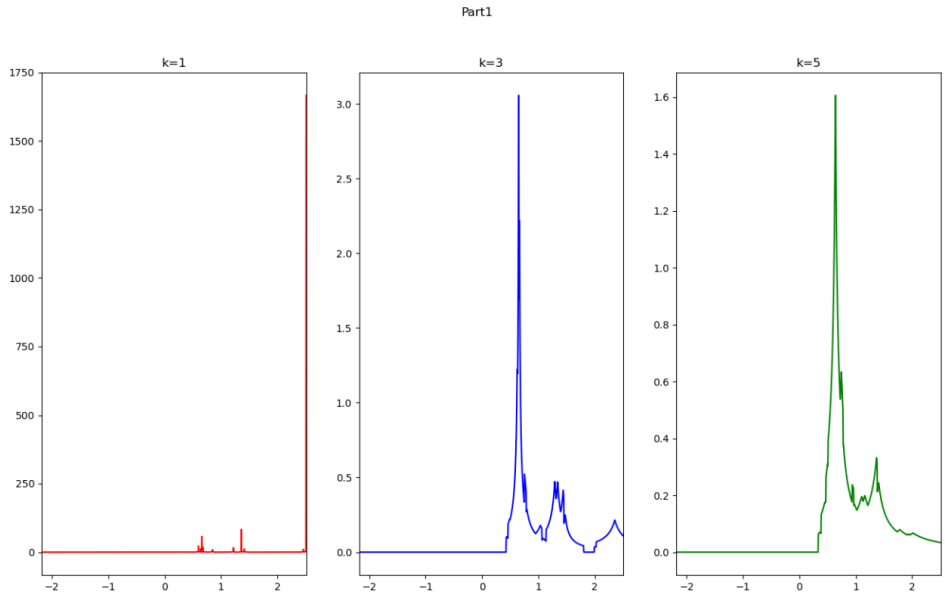
```
16
17      plt.figure(figsize=(16, 9))
18      plt.subplot(131)
19      plt.plot(X_test, y1_test, 'r')
20      plt.xlim([x_min, x_max])
21      plt.title('k=1')
22
23      plt.subplot(132)
24      plt.plot(X_test, y3_test, 'b')
25      plt.xlim([x_min, x_max])
26      plt.title('k=3')
27
28      plt.subplot(133)
29      plt.plot(X_test, y5_test, 'g')
30      plt.xlim([x_min, x_max])
31      plt.title('k=5')
32
33      plt.suptitle('Part1')
34      plt.show()
```

分别执行后：

对于第一种理解不同k下的类条件概率密度如下：



对于第二种理解不同k下的类条件概率密度如下：



如果将其对应的x轴对其，可以发现理解1的图像更加光滑，因为其在数点的过程中新增一个点增加距离均保持一定的一致性，因此整体图像比较光滑。而对于第二种理解的情况下由于考虑到k个点的范围内不一定存在该类的点，因此会出现数据为0的情况。

而对于k=1的时候，由于会导致距离变为0，因此需要进行设置一个极小的数值来保证其数据输出的合理性。

同样相仿上述对于(1)的操作，对于(2)的操作我们仍需先进行数据处理是否摘除，再执行KNN的操作。

对于理解1的数据处理如下：

```
X2 = pd.concat([X['x1'], X['x2']], axis=1)
X_train = X2[y == 2].copy().reset_index(drop=True)
y_train = y[y == 2].copy().reset_index(drop=True)
DealPart2(X_train, y_train)
```

对于理解2的数据处理如下：

```
X2 = pd.concat([X['x1'], X['x2']], axis=1)
X_train = X2.copy().reset_index(drop=True)
y_train = y.copy().reset_index(drop=True)
DealPart2(X_train, y_train)
```

同样两者计算类条件概率密度的函数相同，为 `DealPart2` 如下

```
def DrawPart2(x0, x1, z, k):
    # matplotlib.use('TkAgg')
    # 创建一个图形和两个子图（一个2D，一个3D）
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

    # 第一个子图为3D图
    ax1 = fig.add_subplot(121, projection='3d')
    surf = ax1.plot_surface(x0, x1, z, cmap='viridis')
    ax1.set_xlabel('X1')
    ax1.set_ylabel('X2')
    ax1.set_zlabel('p')
    ax1.set_title(f'k={k}, 3D')
```

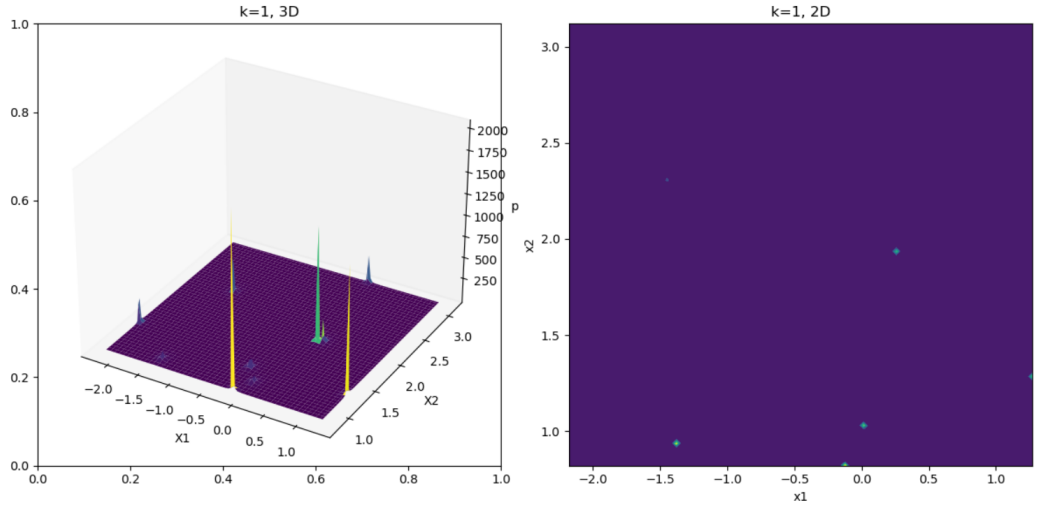
```
    ax1.patch.set_visible(False)
    ax1.grid(False)

    # 第二个子图为2D等高线图
    contour = ax2.contourf(x0, x1, z, cmap='viridis')
    ax2.set_xlabel('x1')
    ax2.set_ylabel('x2')
    ax2.set_title(f'k={k}, 2D')
    # 调整布局
    plt.tight_layout()
    # 显示图形
    plt.show()

def DealPart2(X_train, y_train):
    x1_min = X_train['x1'].min()
    x1_max = X_train['x1'].max()
    x2_min = X_train['x2'].min()
    x2_max = X_train['x2'].max()
    x0, x1 = np.meshgrid(np.linspace(x1_min, x1_max, 100).reshape(-1, 1),
                         np.linspace(x2_min, x2_max, 100).reshape(-1, 1))
    X_test = np.c_[x0.ravel(), x1.ravel()]
    X_train = X_train.to_numpy()
    knn = kNNClassifier()
    y1_post = knn.density(X_train, y_train, X_test, 1)
    y3_post = knn.density(X_train, y_train, X_test, 3)
    y5_post = knn.density(X_train, y_train, X_test, 5)

    # 获得概率密度函数
    y1_density = knn.density(X_train, y_train, X_test, 1)

    y3_density = knn.density(X_train, y_train, X_test, 3)
    y5_density = knn.density(X_train, y_train, X_test, 5)

    y1_test = np.array([d[2] for d in y1_density]).reshape(-1, 1)
    y3_test = np.array([d[2] for d in y3_density]).reshape(-1, 1)
    y5_test = np.array([d[2] for d in y5_density]).reshape(-1, 1)

    DrawPart2(x0, x1, y1_test.reshape(x0.shape), 1)
    DrawPart2(x0, x1, y3_test.reshape(x0.shape), 3)
    DrawPart2(x0, x1, y5_test.reshape(x0.shape), 5)
```

其运行的结果如下：

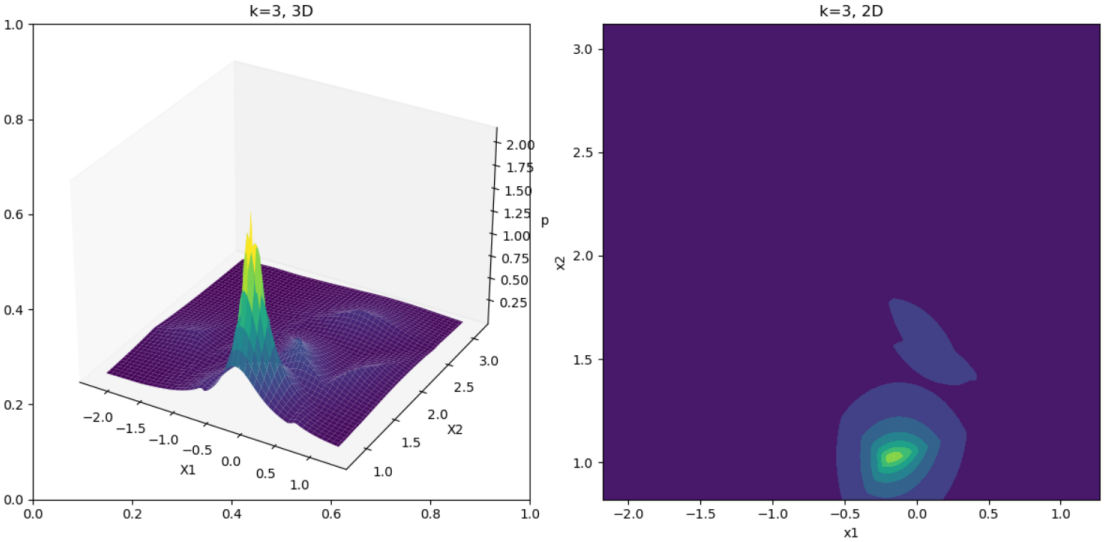
左侧为二维曲面图，右侧为对应的等高线

第一种理解：

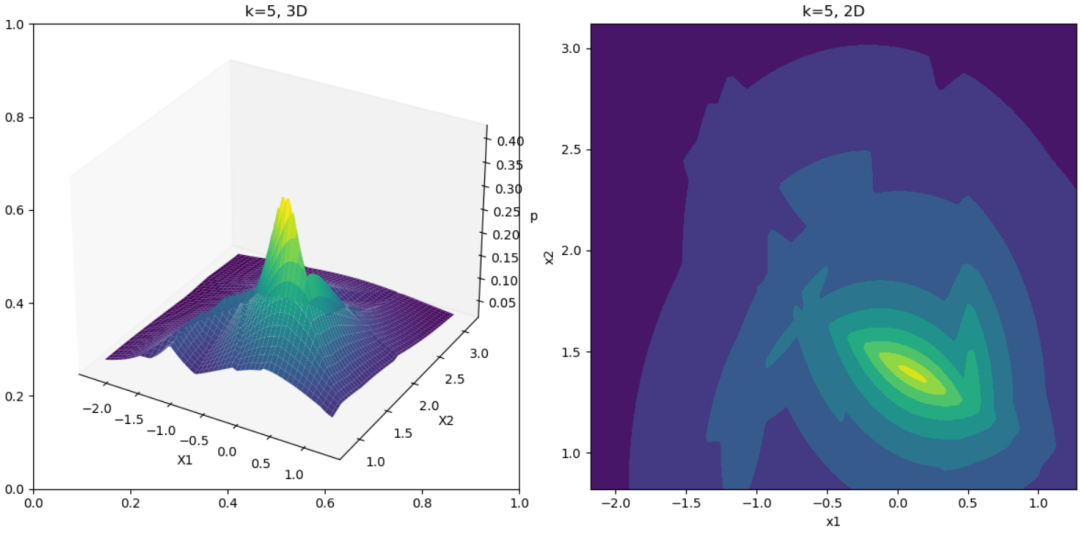
k=1



k=3

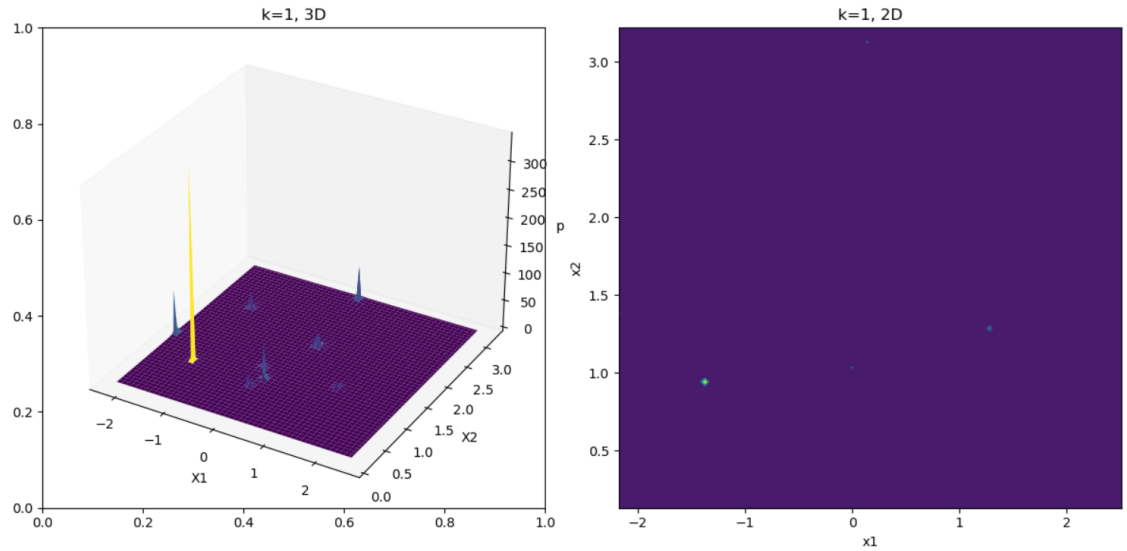


k=5

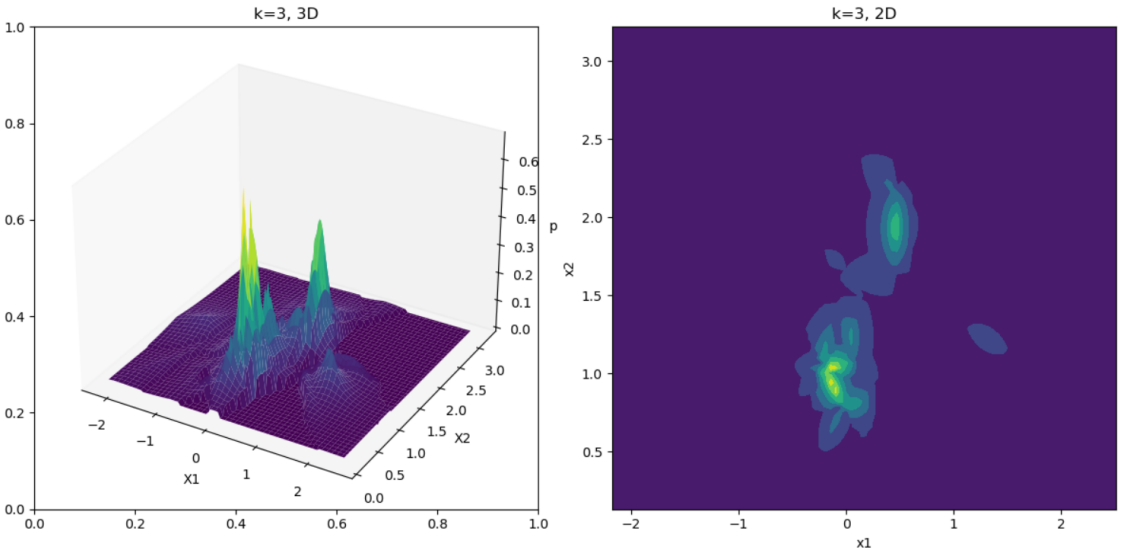


第二种理解：

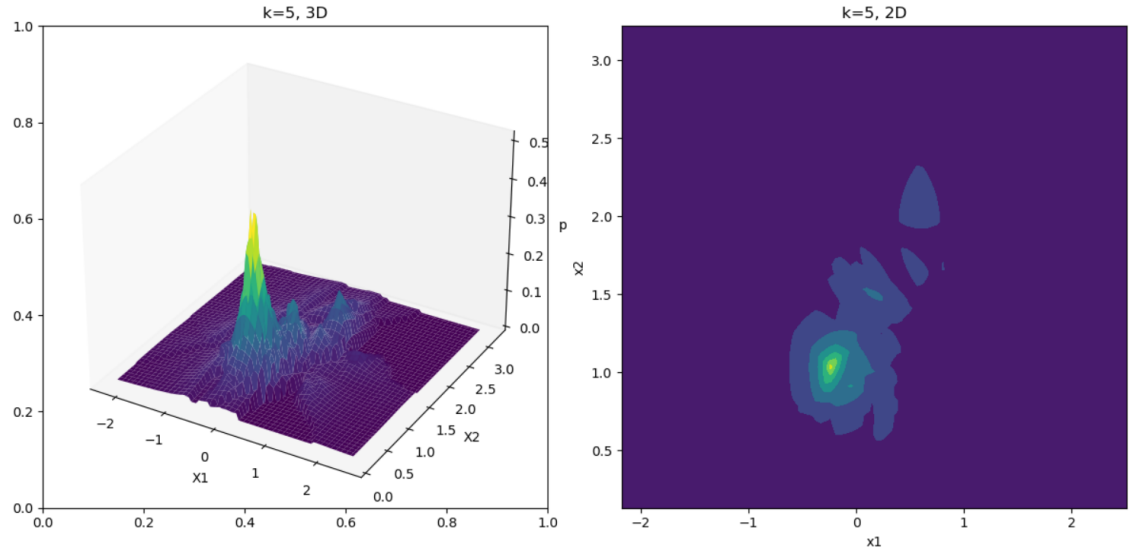
k=1



k=3



k=5



而对于(3) 采用不同的k进行执行，获得其对应的类条件概率密度即可，执行如下代码，得到如下结果

```
knn = kNNClassifier()
for k in range(5):
    density = knn.density(X, y, Xtest, k + 1,2)
    print(f'k={k + 1} 时的概率密度')
    for index in range(len(density)):
        print(density[index])
```

```
k=1 时的概率密度
{1: 0.01883296229251353, 2: 0.22167283203251173, 3: 0.07712096076645929}
{1: 0.04131910919281311, 2: 0.0016351496179347956, 3: 0.0030716239012411896}
{1: 0.1771613350916689, 2: 0.13534049221543457, 3: 0.013723919694781556}
k=2 时的概率密度
{1: 0.03178787537795644, 2: 0.16427006941590264, 3: 0.14466794867681504}
{1: 0.026792930905745034, 2: 0.003220749147742275, 3: 0.005967511370200231}
{1: 0.22110756108533755, 2: 0.26802033748210696, 3: 0.026840350461291625}
k=3 时的概率密度
{1: 0.00879596401261926, 2: 0.23162580018270137, 3: 0.15009942277353658}
{1: 0.017964386176866163, 2: 0.0040412635615593925, 3: 0.008591866296713964}
{1: 0.01102829558991302, 2: 0.3638189115792198, 3: 0.0354162032665907}
k=4 时的概率密度
{1: 0.010231864737074553, 2: 0.23905049106054166, 3: 0.1874814465698455}
{1: 0.013391533141938477, 2: 0.005348468289115052, 3: 0.011381430132195194}
{1: 0.00919145633055199, 2: 0.3362516370263627, 3: 0.040106087762924514}
k=5 时的概率密度
{1: 0.010953452694776046, 2: 0.1393888918855546, 3: 0.11134270190658971}
{1: 0.004295766694553839, 2: 0.006489497138077377, 3: 0.010832627307478103}
{1: 0.007823205569486222, 2: 0.12488416706442222, 3: 0.039732878044842136}
```

4 KNN实战

4-1 实验目的

掌握KNN算法的使用。

一、数据预处理

1.将e2.txt中的数据处理成可以输入给模型的格式

2.是否还需要对特征值进行归一化处理？目的是什么？

二、数据可视化分析

将预处理好的数据以散点图的形式进行可视化，通过直观感觉总结规律，感受 KNN 模型思想与人类经验的相似之处。

三、构建 KNN 模型并测试

1.输出测试集各样本的预测标签和真实标签，并计算模型准确率。

2.选择哪种距离更好？欧氏还是马氏？

3.改变数据集的划分以及 k 的值，观察模型准确率随之的变化情况。

注意：选择训练集与测试集的随机性

四、使用模型构建可用系统

利用构建好的 KNN 模型实现系统，输入为新的数据的三个特征，输出为预测的类别。

4-2 实验过程

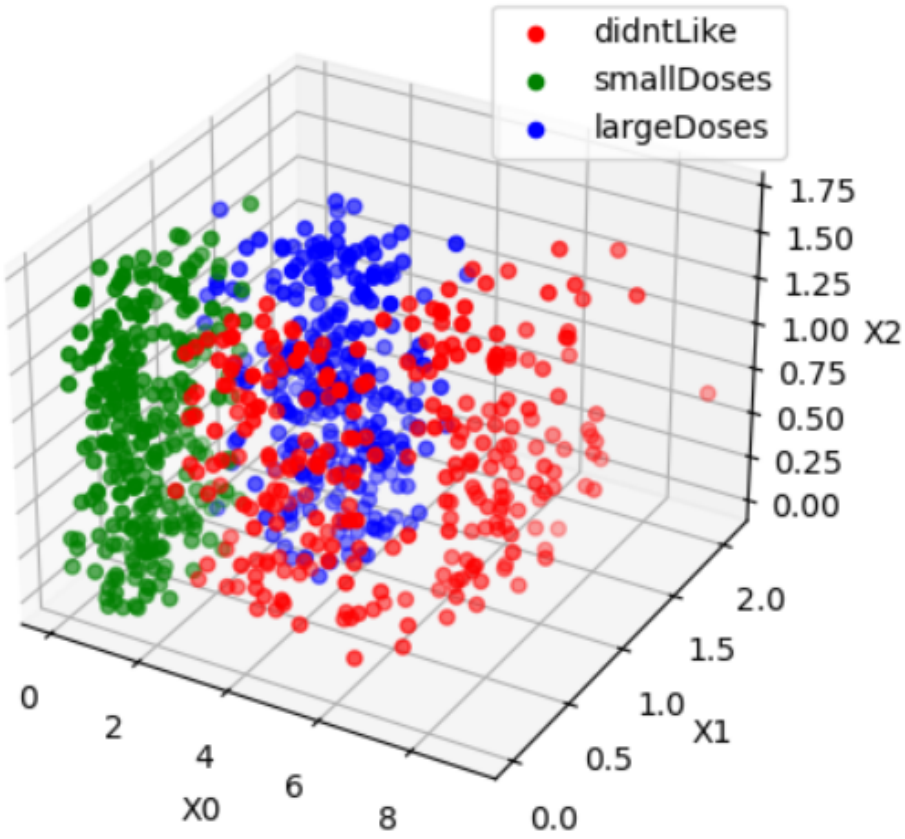
首先获得该数据，利用`pandas`的`read_csv`函数进行读入，通过对数据的观察，对数据进行一个暴力的归一化处理以及标签的数据化。如下：

```
yMapping = {'didntLike': 0, 'smallDoses': 1, 'largeDoses': 2}
yList = ['didntLike', 'smallDoses', 'largeDoses']


def dealForData(data):
    data['x0'] = data['x0'] / 10000
    data['x1'] = data['x1'] / 10
    data['y'] = data['y'].replace(yMapping)
    return data

data = pd.read_csv('../../data/e2.txt', sep="\t", names=['x0', 'x1', 'x2', 'y'])
data = dealForData(data)
```

随后简单对数据集进行一个4：1的划分，并对训练集绘制散点图来可视化

```
def DrawScatterPlot(X, y):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    # 为每个分类的点设置不同的颜色和标签
    colors = ['red', 'green', 'blue']
    labels = ['didntLike', 'smallDoses', 'largeDoses']
    for i in range(3):
        subset = X[y == i]
        ax.scatter(subset['x0'], subset['x1'], subset['x2'], c=colors[i], label=labels[i])
    # 设置图例和坐标轴标签
    ax.legend()
    ax.set_xlabel('X0')
    ax.set_ylabel('X1')
    ax.set_zlabel('X2')
    plt.show()

def splitForData(data, test_size, random_state):
    np.random.seed(random_state)
    data_shuffled = data.sample(frac=1).reset_index(drop=True)
    train_size = int((1 - test_size) * len(data))
    train_data = data_shuffled[:train_size]
    test_data = data_shuffled[train_size:]
    X_train = train_data.drop('y', axis=1)
    y_train = train_data['y']
    X_test = test_data.drop('y', axis=1)
    y_test = test_data['y'].copy()
    return X_train, y_train, X_test, y_test

X_train, y_train, X_test, y_test = splitForData(data, 0.2, 7)
y_test = y_test.to_numpy()
DrawScatterPlot(X_train,y_train)
```

即可获得如右图的散点图：



随后即可执行KNN算法，在上述给定的代码中，我采用可调节的参数来进行选择不同的距离计算方式以及不同的计算概率密度的方式，并在预测中再乘统计的先验概率来进行预测，比单纯采用类条件概率密度的预测在数据不稳定打乱的情况下更好。KNN类的具体描述如下。

```
class kNNClassifier:

    # 该方法基于上述第三问题的KNN中的第一种理解，即将类别分为不同的类进行，不同类比之间互不干涉，但是该方法存在某一类的点少于k个，则会出现概念问题，并且不易解决，因此采用该方法时k不应该选取太大。
    def KNN_Vn(self, X, k, t):
        '''
        基于 p_{n}(x) = k_{n}/(V_{n}*n) 来计算概率密度
        :param X: 原始数据集的某一类别
        :param k: KNN 的k
        :param t: 测试集
        :return: 一个概率密度函数
        '''
        res = np.zeros((t.shape[0],))
        for i, test in enumerate(t):
            dist = np.linalg.norm(X - test, axis=1)
            dist = np.sort(dist)
            if dist[k - 1] != 0:
                res[i] = k / (X.shape[0] * dist[k - 1] ** X.shape[1])
            else:
                res[i] = k / (X.shape[0] * (1e-5) ** X.shape[1])
        return res

```

```
    # 该方法与上述的方法为配套使用的方法，来实现上述描述的理解。
     def kNN_Vn(self, X_train, y_train, X_test, k):
         res = [{} for _ in range(X_test.shape[0])]
         yValueList = y_train.unique().tolist()
         for y in yValueList:
             trainSet = X_train[y_train == y].copy()
             yRes = self.KNN_Vn(trainSet, k, X_test)
             for i in range(len(yRes)):
                 # 为 res 中的每一行的字典添加键值对，键为 y，值为 yRes 中对应的元素
                 res[i][y] = yRes[i]
         return res

    # 计算超球体系数中的gamma函数
     def gamma(self, x):
         # 递归利用已知的Gamma(1/2) = sqrt(pi)
         if abs(x - 0.5) < 1e-6:
             return math.sqrt(math.pi)
         elif abs(x - 1) < 1e-6:
             return 1
         else:
             return (x - 1) * self.gamma(x - 1)

    # 常规基于欧氏距离数点的计算方式
     def kNN_Euler_Count(self, X_train, y_train, X_test, k):
         '''
         基于欧拉距离的计数方法来计算类条件概率密度
         :param X_train: 训练集
         :param y_train: 训练标签
         :param X_test: 测试集
         :param k: k近邻的k
         :return: 测试集的类条件概率密度
         '''
         res = []
         # 超球体体积系数 $V_n(r) = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2} + 1)}
r^n$ 令 $ alpha = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2} + 1)} $
         pi_power = math.pi ** (X_train.shape[1] / 2)
         gamma_value = self.gamma(((X_train.shape[1] / 2) + 1))
         alpha = ((pi_power) / gamma_value) if gamma_value != 0 else
float('inf')
         for i, test in enumerate(X_test):
             dist = np.linalg.norm(X_train - test, axis=1)
             KNN_indices = np.argsort(dist)[:k]
             KNN_labels = y_train[KNN_indices]
             d = dist[KNN_indices[-1]]
             if d == 0:
                 d = 1e-5
             class_probs = {
                 cls: np.sum(KNN_labels == cls) / (X_train.shape[0] * alpha
* (d ** X_train.shape[1]))
                 for cls in np.unique(y_train)}
             res.append(class_probs)
         return res

    # 马氏距离的计算
     def mahalanobis_distance(self, x, dataset, invCov):
         '''
         计算单个样本与数据集所有样本的马氏距离
         :param x:
```

```
        :param dataset:
        :param invCov:
        :return:
        '''
        dist = np.zeros(dataset.shape[0])
        dataset = dataset.to_numpy()
        for i, data in enumerate(dataset):
            delta = x - data
            dist[i] = np.sqrt(delta.dot(invCov).dot(delta.T))
        return dist

    # 基于马氏距离的KNN
    def kNN_Mahalanobis_Count(self, X_train, y_train, X_test, k):
        '''
        基于马氏距离的计数方法来计算类条件概率密度
        :param X_train: 训练集
        :param y_train: 训练标签
        :param X_test: 测试集
        :param k: k近邻的k
        :return: 测试集的类条件概率密度
        '''
        # 超球体体积系数 $V_n(r) = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2} + 1)}
r^n$ 令 $ alpha = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2} + 1)} $
        pi_power = math.pi ** (X_train.shape[1] / 2)
        gamma_value = self.gamma((X_train.shape[1] / 2) + 1)
        alpha = ((pi_power) / gamma_value) if gamma_value != 0 else
float('inf')
        cov = np.cov(X_train.T)
        invCov = inv(cov)
        res = []
        for i, test in enumerate(X_test):
            dist = self.mahalanobis_distance(test, X_train, invCov)
            KNN_indices = np.argsort(dist)[:k]
            KNN_labels = y_train[KNN_indices]
            d = dist[KNN_indices[-1]]
            if d == 0:
                d = 1e-5
            class_probs = {
                cls: np.sum(KNN_labels == cls) / (X_train.shape[0] * alpha
* (d ** X_train.shape[1]))
                for cls in np.unique(y_train)}
            res.append(class_probs)
        return res

    # predict函数
    def execute(self, X_train, y_train, X_test, k, typ=0):
        yValueList = y_train.unique().tolist()
        prior_prob = {}
        for y in yValueList:
            prior_prob[y] = X_train[y_train == y].shape[0]
        if typ == 0:
            probs = self.kNN_Euler_Count(X_train, y_train,
X_test.to_numpy(), k)
        elif typ == 1:
            probs = self.kNN_Mahalanobis_Count(X_train, y_train,
X_test.to_numpy(), k)
        elif typ == 2:
            probs = self.kNN_Vn(X_train, y_train, X_test.to_numpy(), k)
```

```
        predict = np.zeros(X_test.shape[0])
        # print(probs)
        for i, class_prob in enumerate(probs):
            max_prob = -1
            for y in class_prob:
                current_prob = class_prob[y] * prior_prob[y]
                if current_prob > max_prob:
                    max_prob = current_prob
                    predict[i] = y
        return predict

    # 单纯计算类条件概率密度函数
     def density(self, X_train, y_train, X_test, k, typ=0):
        yValueList = y_train.unique().tolist()
        prior_prob = {}
        if not isinstance(X_test, np.ndarray):
            # 如果不是，转换它为numpy数组
            X_test = np.array(X_test)
        for y in yValueList:
            prior_prob[y] = X_train[y_train == y].shape[0]
        if typ == 0:
            probs = self.kNN_Euler_Count(X_train, y_train, X_test, k)
        elif typ == 1:
            probs = self.kNN_Mahalanobis_Count(X_train, y_train, X_test, k)
        elif typ == 2:
            probs = self.kNN_Vn(X_train, y_train, X_test, k)
        return probs
```

随后在随机种子，k保持不变的情况下，在`execute`方法中执行不同的`typ`即可返回不同的预测结果。

```
# 基于一般欧氏距离方法
y_pred = knn.execute(X_train, y_train, X_test,5,0)
accuracy = np.mean(y_pred == y_test)
print(f'一般欧式-Accuracy: {accuracy}')

# 基于马氏距离方法
y_pred = knn.execute(X_train, y_train, X_test,5,1)
accuracy = np.mean(y_pred == y_test)
print(f'一般马氏-Accuracy: {accuracy}')

# 基于单类欧式距离的方法
y_pred = knn.execute(X_train, y_train, X_test,5,2)
accuracy = np.mean(y_pred == y_test)
print(f'独类欧式-Accuracy: {accuracy}')
```

在数据集合划分为随机种子为7时，结果如下：

```
一般欧式-Accuracy: 0.995
一般马氏-Accuracy: 0.965
独类欧式-Accuracy: 0.98
```

可以发现此时采用欧式距离的准确率大于马氏距离。

再采用不同的随机种子，例如21，42，56

当randstate=21时

```
一般欧式-Accuracy: 0.98
一般马氏-Accuracy: 0.95
独类欧式-Accuracy: 0.985
```

当randstate=42时

```
一般欧式-Accuracy: 0.97
一般马氏-Accuracy: 0.96
独类欧式-Accuracy: 0.975
```

当randstate=56时

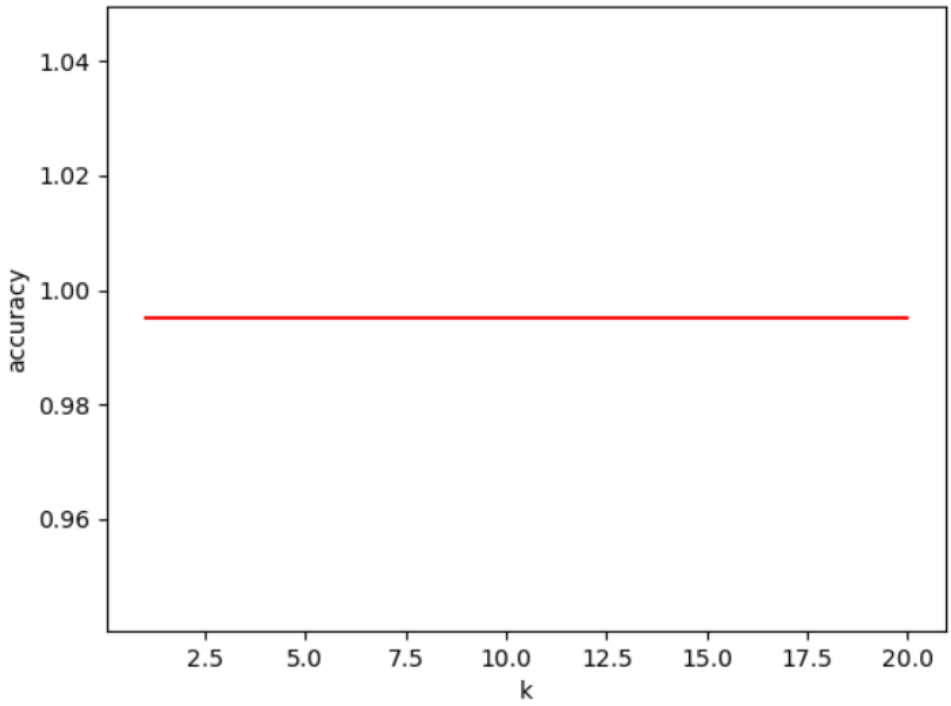
```
一般欧式-Accuracy: 0.945
一般马氏-Accuracy: 0.94
独类欧式-Accuracy: 0.945
```

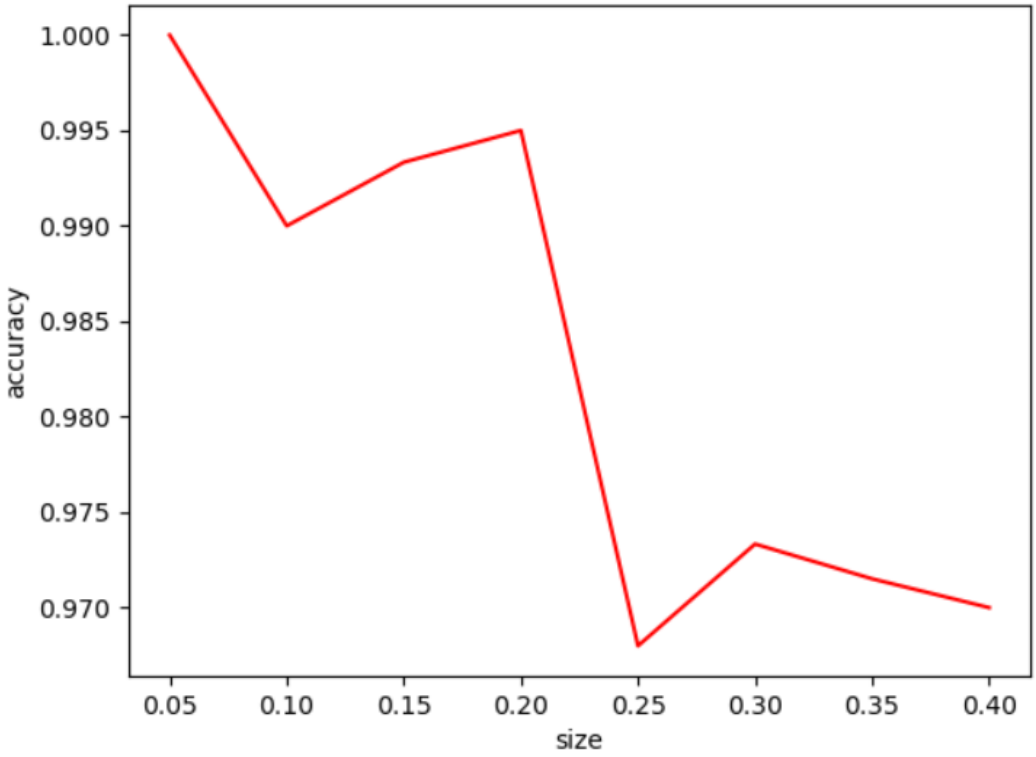
总的来看，欧式距离应当优于马氏距离在该问题上。

随后执行这两个函数来绘制准确率根据不同的划分（测试集占比）和不同k的变化成都

```
def plotAccuracyK(X_train, y_train, X_test, y_test):
    x = np.linspace(1,20,20)
    accuracy = np.zeros(20)
    for k in range(20):
        knn = kNNClassifier()
        y_pred = knn.execute(X_train, y_train, X_test,5, 0)
        accuracy[k]=np.mean(y_pred == y_test)
    plt.plot(x, accuracy,'r-')
    plt.xlabel('k')
    plt.ylabel('accuracy')
    plt.show()

def plotAccuracySize(data,random_state=42):
    x = np.linspace(0.05,0.4,8)
    accuracy = np.zeros(8)
    for i,size in enumerate(x):
        X_train, y_train, X_test, y_test = splitForData(data, size, 7)
        y_test = y_test.to_numpy()
        knn = kNNClassifier()
        y_pred = knn.execute(X_train, y_train, X_test,5, 0)
        accuracy[i] = np.mean(y_pred == y_test)
    plt.plot(x, accuracy, 'r-')
    plt.xlabel('size')
    plt.ylabel('accuracy')
    plt.show()

plotAccuracyK(X_train, y_train, X_test, y_test)
plotAccuracySize(data)
```

得到如下结果：





可以发现随着k的增长，KNN的准确率变化不大在该数据集下。

但随着测试集占比越来越大，KNN的准确率逐步下降。

随后输入这样的数据，查看其是否可以预测

```
Temp = [40000,8,0.9]
Temp = pd.DataFrame([Temp], columns=['x0', 'x1', 'x2'])
y_pred = knn.execute(X_train, y_train, Temp,5,1)
index = int(y_pred[0])  # 转换为整数
print(yList[index])
```

其输出 `didntLike` 即可以正常进行预测。

实验到此结束。

代码结构

```
/Exp2/
|------- code/
|        |------- Part1 第一部分的代码
|        |        |------- DataProcessor.py 数据处理的工具类
|        |        |------- exp2-1.py 线性分类主函数代码
|        |        |------- LinearClassifier.py 线性分类类
|        |------- Part2 第二部分的代码
|        |        |------- exp2-2.py 最大似然主函数代码
|        |------- Part3 第三部分的代码
|        |        |------- exp2-3-kNearestNeighbor.py KNN执行主代码
|        |        |------- exp2-3-ParzenWindow.py Parzen窗执行主代码
|        |------- Part4 第四部分代码
|        |        |------- exp2-4.py KNN实战执行主代码
|        |        |------- KNN.py part3和part4有关KNN的KNN类
|
|-------- data/
|        |------- e2data1.mat part1数据
|        |------- exp2-2.xlsx part2数据
|        |------- exp2-3.xlsx part3数据
|        |------- e2.txt part4数据
|
|-------- images/实验报告有关图片
|
|-------- Exp2.md 实验报告Markdown
|
|-------- Exp2.pdf 实验报告pdf
```

心得体会

在这次机器学习的基础实验中，我有幸深入探讨了四个核心部分：简单线性分类模型、有参估计、多维统计分析及无参估计，每个部分都让我获得了宝贵的学术及实践经验。

简单线性分类模型： 通过实现简单的线性分类模型，我不仅复习了线性代数的基本知识，还学习了如何通过编程将理论应用到实际数据分析中。这一部分的挑战在于选择合适的模型参数和理解模型的决策边界。实验过程中，我通过可视化手段直观地观察了分类效果，这极大地增强了我的直观理解和对模型调优的实践能力。

有参估计中的极大似然估计： 在这一部分，我学习了如何在单维和多维情境下使用极大似然估计来计算数据的均值和方差。通过对比单维和多维的计算方法，我更深入地理解了这些统计量在不同维度下的行为和意义，尤其是在处理实际数据集时如何应用这些理论来提取有用的统计信息。

无参估计的Parzen窗和KNN算法： 探索无参估计的方法开阔了我的视野，特别是在使用Parzen窗和KNN算法进行数据分类和回归分析方面。我实践了如何根据数据的分布选择合适的窗口大小和邻居数，这对于优化模型性能至关重要。通过实际操作，我学会了调整这些参数以适应不同的数据集，进而优化分类和预测的准确性。

整体反思： 这四个实验部分使我认识到，无论是有参还是无参估计，理解其背后的数学原理和如何将这些原理应用于实际问题都是至关重要的。此外，实验不仅提高了我的编程技能，还增强了我对机器学习模型如何在现实世界中应用的认识。我对未来在更复杂数据集上应用这些技术感到兴奋，并期待在未来的学习和研究中继续探索更多机器学习的领域。