西北大学

智能信息系统综合实践

实验报告

题　目:	**第四次作业**
年　级:	2020
专　业:	软件工程
姓　名:	刘航

一、 题目（原题目）

作业1： 数据集内包含 3 类共 150 条记录，每类各 50 个数据，每条记录都有 4 项特征：花萼长度、花萼宽度、花瓣长度、花瓣宽度，可以通过这4个特征预测鸢尾花卉属于（iris-setosa，iris-versicolour，iris-virginica）中的哪一品种。

***要求：**

① 在鸢尾花数据集上，尽量手动编写朴素贝叶斯分类代码，完成分类实验。（可参考李航统计学习方法）

② 在①的基础上，对鸢尾花中特征部分进行划分，分别利用1个，2个，3个，4个特征进行分类，比较不同特征对分类结果的影响，画图比较。

作业2： 选用的MNIST数据集，每条记录都有 28*28 项特征，随机选择70%训练，30%测试。

***要求：**

① 在数据集上，利用朴素贝叶斯分类代码，完成分类实验，并画图给出不同数字的准确率，并尝试分析其原因。

② 利用PCA将28*28维度降维到某个维度（如10，20等），完成分类并计算其准确率。

③ 贝叶斯中唯一的参数—平滑系统，可尝试调整参数，比较其对分类的影响。

二、 解题步骤（思路 + 代码）

问题一：编写朴素贝叶斯分类代码。

朴素贝叶斯原理分析。

- 假设有 N 种可能的类别标记，即 $y = \{c_1, c_2, \ldots, c_N\}$ ，λ_{ij} 是将一个真实标记为 c_j 的样本误分类为 c_i 所产生的损失。基于后验概率 $P(c_i \mid \mathbf{x})$ 可获得将样本 $\mathbf{x}$ 分类为 c_i 所产生的期望损失（expected loss），即在样本上的“条件风险”（conditional risk）

$$R(c_i \mid \mathbf{x}) = \sum_{j=1}^{N} \lambda_{ij} P(c_j \mid \mathbf{x}) \qquad (7.1)$$

- 我们的任务是寻找一个判定准则 $h : X \mapsto Y$ 以最小化总体风险

$$R(h) = \mathbf{E}_x \left[R(h(\mathbf{x}) \mid \mathbf{x}) \right] \qquad (7.2)$$

- 显然，对每个样本 $\mathbf{x}$，若 h 能最小化条件风险 $R(h(\mathbf{x}) \mid \mathbf{x})$ ，则总体风险 $R(h)$ 也将被最小化。
- 这就产生了贝叶斯判定准则（Bayes decision rule）： 为最小化总体风险，只需在每个样本上选择那个能使条件风险 $R(c \mid \mathbf{x})$ 最小的类别标记，即

$$h^*(x) = \underset{c \in y}{\operatorname{argmin}} R(c \mid x) \qquad (7.3)$$

  - 此时，被称为贝叶斯最优分类器(Bayes optimal classifier)，与之对应的总体风险 $R(h^*)$ 称为贝叶斯风险 (Bayes risk)
  - $1 - R(h^*)$ 反映了分类起所能达到的最好性能，即通过机器学习所能产生的模型精度的理论上限。

- 基于属性条件独立性假设，(7.8)可重写为

$$P(c \mid \mathbf{x}) = \frac{P(c)P(\mathbf{x} \mid c)}{P(\mathbf{x})} = \frac{P(c)}{P(\mathbf{x})} \prod_{i=1}^{d} P(x_i \mid c) \qquad (7.14)$$

  - 其中 d 为属性数目，x_i 为 $\mathbf{x}$ 在第 i 个属性上的取值。

由于对所有类别来说 $P(x)$ 相同，因此基于式 (7.6)的贝叶斯判定准则有

$$h_{nb}(\mathbf{x}) = \underset{c \in y}{\operatorname{argmax}} P(c) \prod_{i=1}^{d} P(x_i \mid c) \qquad (7.15)$$

显然, 朴素贝叶斯分类器的训练过程就是基于训练集 D 来估计类先验概率 $P(c)$, 并为每个属性估计条件概率 $P(x_i \mid c)$.

令 D_c 表示训练集 D 中第 c 类样本组成的集合, 若有充足的独立同分布样本, 则可容易地估计出类先验概率

$$P(c) = \frac{|D_c|}{|D|} \,. \tag{7.16}$$

对离散属性而言, 令 D_{c,x_i} 表示 D_c 中在第 i 个属性上取值为 x_i 的样本组成的集合, 则条件概率 $P(x_i \mid c)$ 可估计为

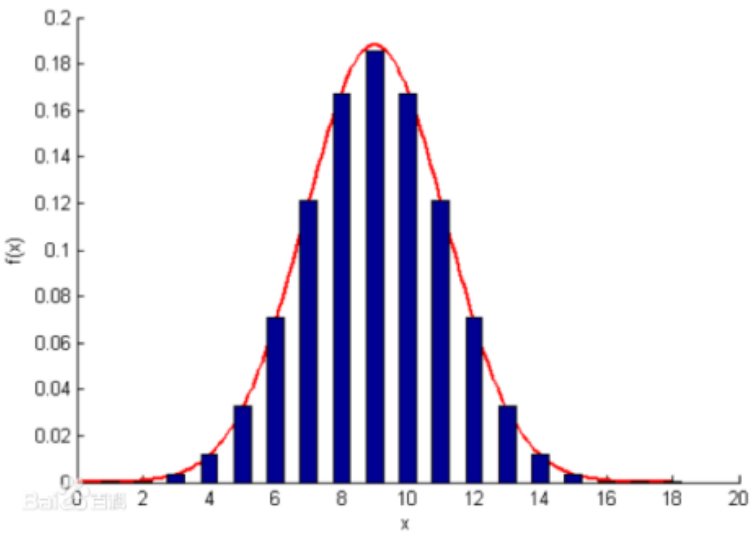
$$P(x_i \mid c) = \frac{|D_{c,x_i}|}{|D_c|} \,. \tag{7.17}$$

对连续属性可考虑概率密度函数, 假定 $p(x_i \mid c) \sim \mathcal{N}(\mu_{c,i}, \sigma_{c,i}^2)$, 其中 $\mu_{c,i}$ 和 $\sigma_{c,i}^2$ 分别是第 c 类样本在第 i 个属性上取值的均值和方差, 则有

$$p(x_i \mid c) = \frac{1}{\sqrt{2\pi}\sigma_{c,i}} \exp\left(-\frac{(x_i - \mu_{c,i})^2}{2\sigma_{c,i}^2}\right) \,. \tag{7.18}$$

处理连续数值问题的另一种常用的技术是通过离散化连续数值的方法，通常，当训练样本数量较少或者是精确的分布已知时，通过概率分布的方法是一种更好的选择。

在大量样本的情形下离散化的方法表现更优，因为大量的样本可以学习到数据的分布。由于朴素贝叶斯是一种典型的用到大量样本的方法（越大计算量的模型可以产生越高的分类精确度），所以朴素贝叶斯方法都用到离散化方法，而不是概率分布估计的方法。



为了避免其他属性携带的信息被训练集中未出现的属性值“抹去”, 在估计概率值时通常要进行“平滑”(smoothing), 常用“拉普拉斯修正”(Laplacian correction). 具体来说, 令 N 表示训练集 D 中可能的类别数, N_i 表示第 i 个属性可能的取值数, 则式(7.16)和(7.17)分别修正为

$$\hat{P}(c) = \frac{|D_c| + 1}{|D| + N} \,, \tag{7.19}$$

$$\hat{P}(x_i \mid c) = \frac{|D_{c,x_i}| + 1}{|D_c| + N_i} \,. \tag{7.20}$$

对于少量样本，如果特征的取值近似符合正态分布，可以使用高斯贝叶斯函数进行计算。

如果和正态分布的差别较大，那么需要将特征取值进行离散化。

实验步骤：

数据分析：

原始数据集合中有4个特征，150个数据。这些特征的取值是离散的。

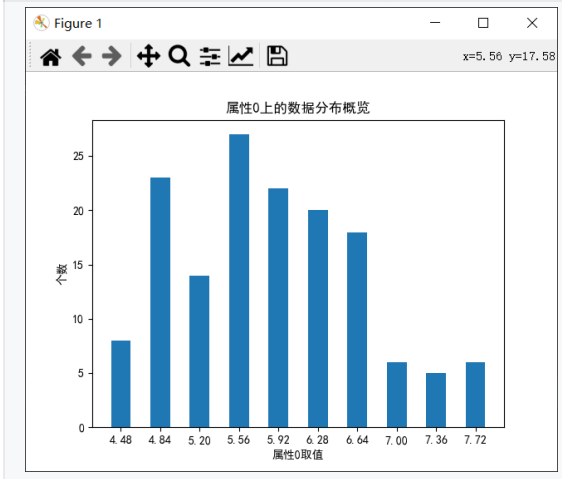
分析数据，如果数据分布大体符合正态分布，那么可以考虑使用高斯贝叶斯分类器。

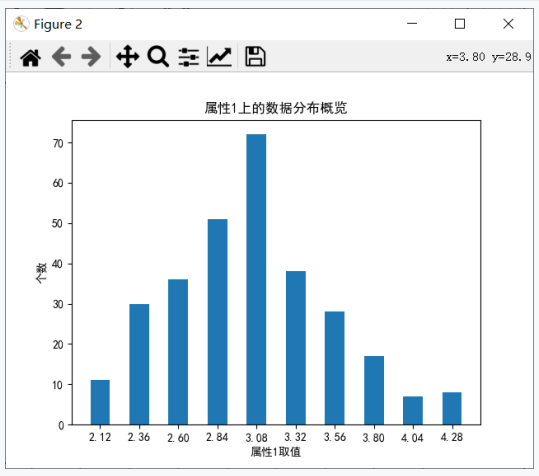
如果数据分布和高斯分布相差较大，那么使用高斯分布的准确率可能会很低。

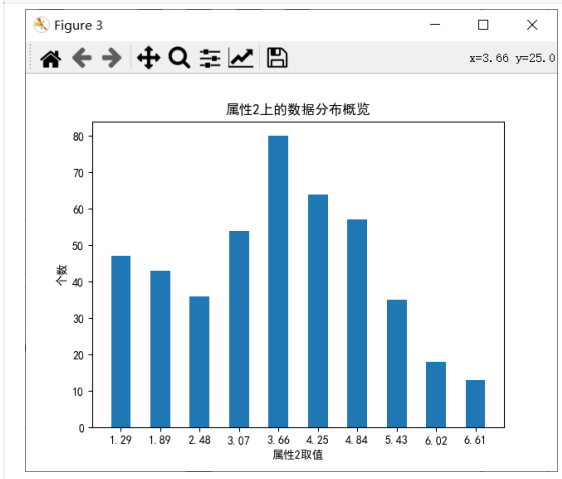
需要将连续数据进行离散化，然后使用多项式的贝叶斯分类器。

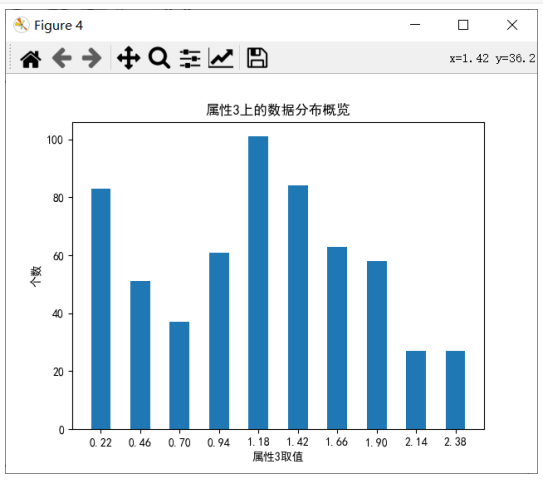
首先观察数据集的4个特征的数据分布情况。

可以看出它们都大致符合正态分布的情况。可以近似看作正态分布处理









代码说明：

step1：代码编写前的分析

题中的数据集的特征都是连续的，并且大致符合正态分布的规律，对于题中要求的分类任务，可以假设每个特征的数据分布是正态分布，建立高斯贝叶斯模型。

step2：需要记录的数据和存储形式。

结合公式分析预测任务需要的数据。

令 D_c 表示训练集 D 中第 c 类样本组成的集合，若有充足的独立同分布样本，则可容易地估计出类先验概率

$$P(c) = \frac{|D_c|}{|D|} . \tag{7.16}$$

对连续属性可考虑概率密度函数，假定 $p(x_i \mid c) \sim \mathcal{N}(\mu_{c,i}, \sigma_{c,i}^2)$，其中 $\mu_{c,i}$ 和 $\sigma_{c,i}^2$ 分别是第 c 类样本在第 i 个属性上取值的均值和方差，则有

$$p(x_i \mid c) = \frac{1}{\sqrt{2\pi}\sigma_{c,i}} \exp\left(-\frac{(x_i - \mu_{c,i})^2}{2\sigma_{c,i}^2}\right) . \tag{7.18}$$

（1）进行一个样本类别估计时，需要知道训练集合的样本数量，以及训练集合的每一类的样本数量。这是计算公式中需要的类先验概率的条件。

（2）连续属性假设这个属性的取值符合正态分布，需要知道这个正态分布的均值和方差。这里需要计算出每个类中的每个特征的均值和反差。

一个高斯贝叶斯分类器进行预测的数据就是上面这些。

现在进行创建一个高斯贝叶斯分类器模型，存储上面所有的数据。

考虑到类别标签不一定都是数字形式，可以为字符串这些。

可以先将每个类的样本个数，每个类的每个特征的均值和反差存储为一个字典的样式。

```
def Create_GaussianNB_classifier(train_x,train_y,smooth = 1):
    label_dict = {}
    label_feature_dict = {}
    for label in train_y:
        label_dict[label] = label_dict.get(label, 0) + 1

    for label in label_dict:
        # 这里train_x和train_y的数据类型要求是array类型的
        train_x_label = train_x[train_y == label]
        # label_feature_dict[label] = [0] * np.size(train_x, 1)
        label_feature_dict[label] = np.array([np.mean(train_x_label, axis=0), np.var(train
_x_label, axis=0)])
    # print('开始构造贝叶斯')
    # print(dict)
    # print(dict_att_2)
    b_classfy = GaussianNB(label_dict, label_feature_dict, smooth)

    # print(label_dict)
    # print(label_feature_dict)
    # print('退出')
    return b_classfy

    pass
```

step3：进行预测函数的编写。

通过上面步骤存储的数据，结合公式计算一个样本属于每个类的概率。

找出概率最大的类别作为预测类别。

需要一个数组存储每个样本的属于每个类别的概率。用来查找最大概率所属类的可能。

经过分析和验证这里最好可以结合矩阵乘法进行计算。

python库中的矩阵乘法进行了优化，使用矩阵乘法的效率经过验证可以提高百倍左右。

```
class GaussianNB:

    def __init__(self,c_dict = {},att_dict = {},smooth = 1):

        # 存放类标的字典，里面是训练集合中某一类的个数
        self.c_dict = c_dict
        # 类和特征的字典
        self.att_dict = att_dict
        # self.att_type = att_type
        self.smooth = smooth
    def predicate(self,test_x):

        train_num = 0
        for item in self.c_dict:
            train_num += self.c_dict[item]

        # n*10矩阵,存放的是测试集合属于每一类的概率
        x_label_p = np.zeros((len(test_x),len(self.c_dict)))

        # 存放的是标签的集合，即上面的列的标签10个
        label_set = np.zeros(len(self.c_dict),dtype=int)
        num = 0
        for label in self.c_dict:
            pt = (self.c_dict[label]+self.smooth)/(train_num+self.smooth*len(self.c_dict))
```

```
            u = self.att_dict[label][0]
            sigma = self.att_dict[label][1]
            sigma[sigma==0] = 1e-4
            p_c_x = np.exp(-(test_x-u)**2/(2 * sigma)) / (np.sqrt(2*np.pi*sigma))

            t = 1
            for j in range(np.size(p_c_x,1)):
                t *= p_c_x[:,j]


            x_label_p[:, num] = pt * t
            label_set[num] = label
            num += 1

        max_index = np.argmax(x_label_p,axis=1)

        predications = label_set[ max_index]
        return predications
```

step4：导入数据，进行训练。

```
def load_data():


    iris_datas = datasets.load_iris()
    x = iris_datas.data
    y = iris_datas.target
    return x,y

```

进行可视化和分析

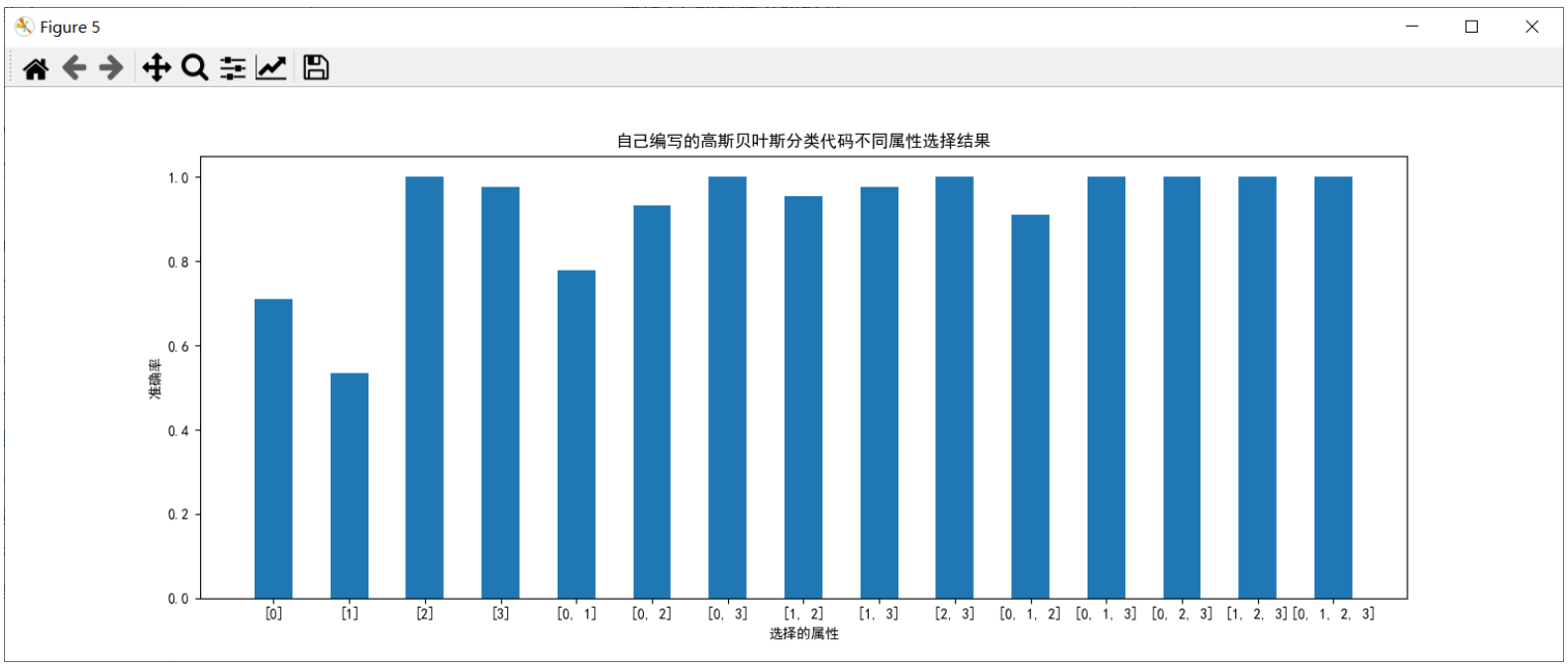
```
def iris_train():

    # x = x[:, 3:4]
    hang = [[0],[1],[2],[3],[0,1],[0,2],[0,3],[1,2],[1,3],[2,3],[0,1,2],[0,1,3],[0,2,3],[1,2,3],[0,1,2,3]]

    x, y = load_data()


    lisna_num = 10
    x_lisan = np.zeros(lisna_num)
    lisna_xlabel = ['']*10
    for i in range(np.size(x,1)):
        min_v = np.min(x[:,i])
        max_v = np.max(x[:,i])
        gap = (max_v-min_v)/lisna_num

        for k in range(len(x)):

            for j in range(lisna_num):
                if x[k][i]>min_v+j*gap and x[k][i]<=min_v+(j+1)*gap:
                    x_lisan[j] += 1

        for ii in range(lisna_num):
            lisna_xlabel[ii] = f'{((min_v+ii*gap) + (min_v+(ii+1)*gap))/2:0.2f}'

        # print(lisna_xlabel)
        # print(x_lisan)
        plt.figure()
        plt.xlabel(f'属性{i}取值')
```

```
        plt.ylabel(f'个数')
        plt.title(f'属性{i}上的数据分布概览')
        plt.bar(lisna_xlabel,x_lisan,0.5)


    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3)
    t_x_train = x_train
    t_x_test = x_test
    predications = np.zeros(len(hang))
    predications_compare = np.zeros(len(hang))
    predications_compare_2 = np.zeros(len(hang))
    x_label = ['']*len(hang)
    for i in range(len(hang)):

        x_label[i] = f'{hang[i]}'
        # x = x[np.linspace(0, len(x),1, dtype=int), hang[i]]
        # x = x[:, hang[i]]
        x_train = t_x_train[:, hang[i]]
        x_test = t_x_test[:,hang[i]]
        # print(x)

        # att_type = np.array([1] * x_train.shape[1])
        b_classfy = Create_GaussianNB_classifier(x_train, y_train, smooth=0)
        y_p = b_classfy.predicate(x_test)
        # print(y_test.tolist())
        gaosi_x = sklearn.naive_bayes.GaussianNB()
        multy_x = sklearn.naive_bayes.MultinomialNB()
        gaosi_x.fit(x_train, y_train)
        multy_x.fit(x_train, y_train)
        y_p_gaosi = gaosi_x.predict(x_test)
        y_p_mulity = multy_x.predict(x_test)
        print('选择属性：',hang[i])
        print('真实值')
        print(y_test)
        print('预测值')
        print(y_p)
        # print(y_test == y_p)
        predications[i] = np.sum(y_test == y_p)/len(y_test)
        predications_compare[i] = np.sum(y_test == y_p_gaosi)/len(y_test)
        predications_compare_2[i] = np.sum(y_test == y_p_mulity)/len(y_test)
        print('准确率',predications[i])
        pass

    plt.rcParams["font.sans-serif"] = "SimHei"
    plt.rcParams["axes.unicode_minus"] = False
    plt.figure()
    plt.bar(x_label,predications,0.5)
    plt.xlabel("选择的属性")
    plt.ylabel("准确率")
    plt.title("自己编写的高斯贝叶斯分类代码不同属性选择结果")
    plt.figure()
    plt.bar(x_label, predications_compare, 0.5)
    plt.xlabel("选择的属性")
    plt.ylabel("准确率")
    plt.title("机器学习库的高斯贝叶斯分类代码不同属性选择结果")

    plt.figure()
    plt.bar(x_label, predications_compare_2, 0.5)
    plt.xlabel("选择的属性")
    plt.ylabel("准确率")
    plt.title("机器学习库的多项式贝叶斯分类代码不同属性选择结果")
    plt.show()
```

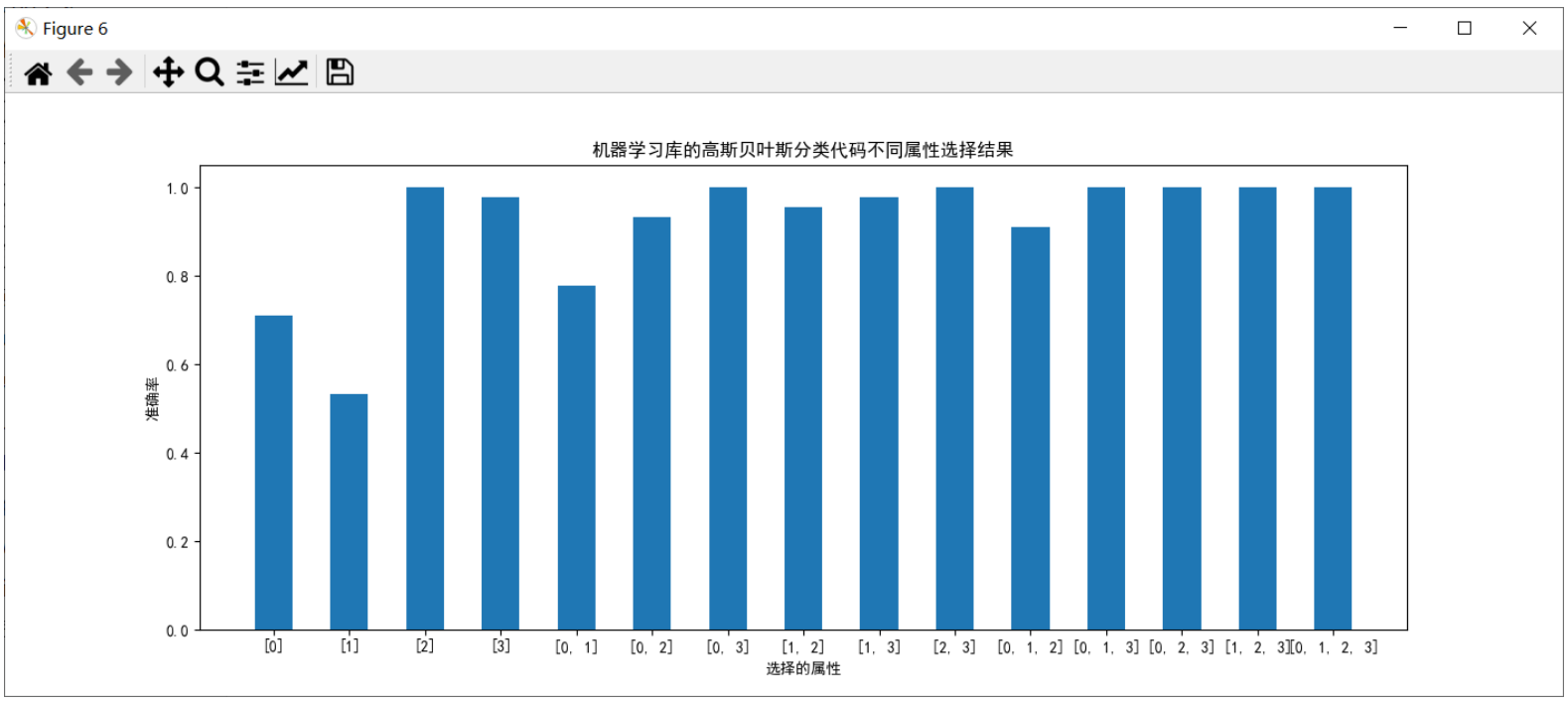
不同特征对分类的影响。

进行各种不同的贝叶斯函数在选取数据集的不同特征进行分类的准确性图表。

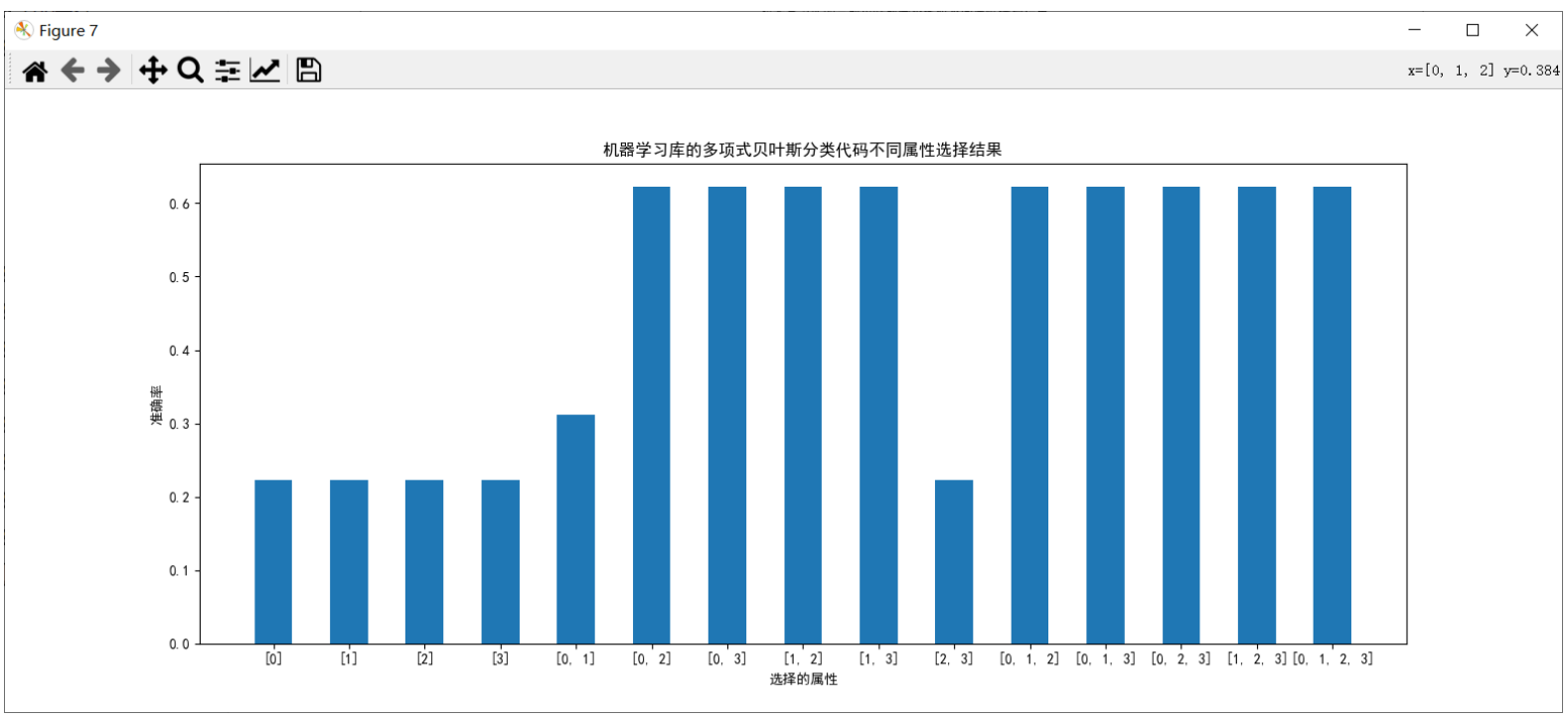
（1）自己手动编写的高斯贝叶斯分类器在不同特征上的分类准确率情况。



（2）机器学习库的高斯贝叶斯分类器在不同特征上的分类准确率情况。



（3）机器学习库的多项式贝叶斯分类器在不同特征上的分类准确率情况。




```
选择属性： [2]
真实值
[1 2 0 2 2 1 0 1 0 0 1 0 0 2 1 0 1 1 0 0 2 0 0 1 2 0 1 2 1 0 0 1 1 1 1 0 0
 0 2 1 2 0 2 1 1]
预测值
[1 2 0 2 2 1 0 1 0 0 1 0 0 2 1 0 1 1 0 0 2 0 0 1 2 0 1 2 1 0 0 1 1 1 1 0 0
 0 2 1 2 0 2 1 1]
准确率 1.0
选择属性： [3]
真实值
[1 2 0 2 2 1 0 1 0 0 1 0 0 2 1 0 1 1 0 0 2 0 0 1 2 0 1 2 1 0 0 1 1 1 1 0 0
 0 2 1 2 0 2 1 1]
预测值
[1 2 0 2 2 1 0 1 0 0 1 0 0 2 1 0 1 1 0 0 2 0 0 1 1 0 1 2 1 0 0 1 1 1 1 0 0
 0 2 1 2 0 2 1 1]
准确率 0.9777777777777777
选择属性： [0, 1]
真实值
[1 2 0 2 2 1 0 1 0 0 1 0 0 2 1 0 1 1 0 0 2 0 0 1 2 0 1 2 1 0 0 1 1 1 1 0 0
 0 2 1 2 0 2 1 1]
预测值
```

```
预测值
[1 2 0 2 2 1 0 1 0 0 1 0 0 2 1 0 1 1 0 0 2 0 0 1 2 0 1 2 1 0 0 1 1 1 1 0 0
 0 2 1 2 0 2 1 1]
准确率 1.0
选择属性： [1, 2, 3]
真实值
[1 2 0 2 2 1 0 1 0 0 1 0 0 2 1 0 1 1 0 0 2 0 0 1 2 0 1 2 1 0 0 1 1 1 1 0 0
 0 2 1 2 0 2 1 1]
预测值
[1 2 0 2 2 1 0 1 0 0 1 0 0 2 1 0 1 1 0 0 2 0 0 1 2 0 1 2 1 0 0 1 1 1 1 0 0
 0 2 1 2 0 2 1 1]
准确率 1.0
选择属性： [0, 1, 2, 3]
真实值
[1 2 0 2 2 1 0 1 0 0 1 0 0 2 1 0 1 1 0 0 2 0 0 1 2 0 1 2 1 0 0 1 1 1 1 0 0
 0 2 1 2 0 2 1 1]
预测值
[1 2 0 2 2 1 0 1 0 0 1 0 0 2 1 0 1 1 0 0 2 0 0 1 2 0 1 2 1 0 0 1 1 1 1 0 0
 0 2 1 2 0 2 1 1]
准确率 1.0
```

分析上面的图表可以看出

（1）2号和3号特征是影响分类准确性的主要特征。并且在仅用2号特征

进行分类的情况下就以及达到非常好的效果了。因此可以推断出，2号特征（花瓣长度）是样本的主要特征。

（2）在这个数据集中特征都是连续特征，使用多项式贝叶斯分类器的效果不如高斯贝叶斯分类器。多项式

分类器不适合在连续特征中。如果要使用最好进行离散化后使用。

问题二

使用上面的贝叶斯分类器在mnist数据集中进行分类。

数据集分析：

（1）mnist数据集合是一个28*28 的手写图片数据，这个数据集合中由6万个训练样本和1万个测试样本。

首先如果要使用上面的分类器对这个数据集进行分类，需要将数据格式转换成一样的才可以。

要将28*28的图片拉伸成1 * 784维的数据格式。

（2）其次拉伸之后的特征数量达到784个，训练样本数量也多达6万个。

如果要使用上面分类器模型可能会出现运行时间过长的问题。可能需要优化算法。

（3）上面的分类器的特征都是连续值，这里因为图片的像素都是离散值，直接使用可能会出现问题。

需要进行数据的处理。

代码分析：

step1：数据的预处理。

首先将28*28的图片转换为 1 * 784的行向量形式。

```
# 将一张图片矩阵拉伸为一个行向量,传入的是x的集合
def imageMatrix_to_hang(x_train):

    temp = np.zeros([np.size(x_train,0),np.size(x_train,1)*np.size(x_train,2)])
    for i in range(np.size(x_train,0)):
        temp[i, :] = x_train[i].reshape(1, -1)
    return temp
    pass
```

step2：加载数据。

```
        (x_train, y_train), (x_test, y_test) = mnist.load_data()

        x_train = imageMatrix_to_hang(x_train)
        x_test = imageMatrix_to_hang(x_test)
```

step3：进行训练和结果展示，与库中的模型进行比较。

```
def minist_train():

    print('数据开始运行')
    start_load_time = time.time()
    (x_train, y_train), (x_test, y_test) = mnist.load_data()

    x_train = imageMatrix_to_hang(x_train)
    x_test = imageMatrix_to_hang(x_test)

    end_load_time = time.time()


    # 构造连续属性的贝叶斯分类
```

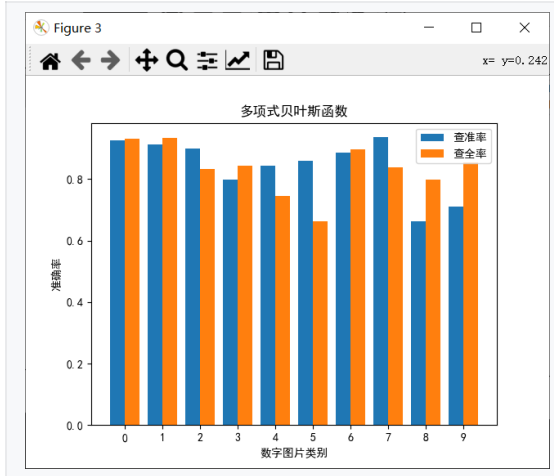
```
    b_classfy = Create_GaussianNB_classifier(x_train, y_train, smooth=0)

    # # 构造离散属性的贝叶斯
    # b_classfy = Create_MultinomialNB_classifier(x_train, y_train, smooth=1)


    end_train_time = time.time()
    y_p = b_classfy.predicate(x_test)

    print('************')
    print('自己编写的函数的准确率')
    print(y_test)
    print(y_p)
    print(np.sum(y_test == y_p))
    print(np.sum(y_test == y_p)/len(y_p))

    end_pridecate_time = time.time()
    # y_p = np.array(y_p)
    gaosi_x = sklearn.naive_bayes.GaussianNB()
    mul_x = sklearn.naive_bayes.MultinomialNB()
    gaosi_x.fit(x_train, y_train)
    mul_x.fit(x_train, y_train)
    y_p_gaosi = gaosi_x.predict(x_test)
    y_p_mul = mul_x.predict(x_test)
    end_ku_time = time.time()

    print('高斯库函数的准确率')
    print(np.sum(y_test == y_p_gaosi))
    print(np.sum(y_test == y_p_gaosi) / len(y_p_gaosi))
    print('多项式库函数的准确率')
    print(np.sum(y_test == y_p_mul) / len(y_p_mul))

    print(f'加载数据和处理耗时:{end_load_time - start_load_time:.2f}s')
    print(f'训练模型耗时:{end_train_time - end_load_time:.2f}s')
    print(f'预测耗时:{end_pridecate_time - end_train_time:.2f}s')
    print(f'调库用的时间:{end_ku_time - end_pridecate_time:.2f}s')
```

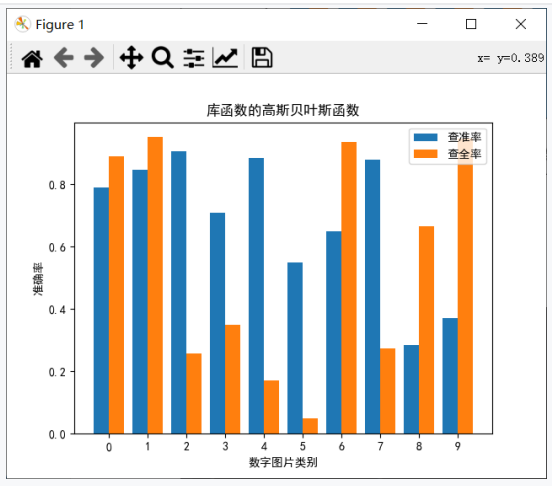
（1）这是没有进行归一化处理的结果。

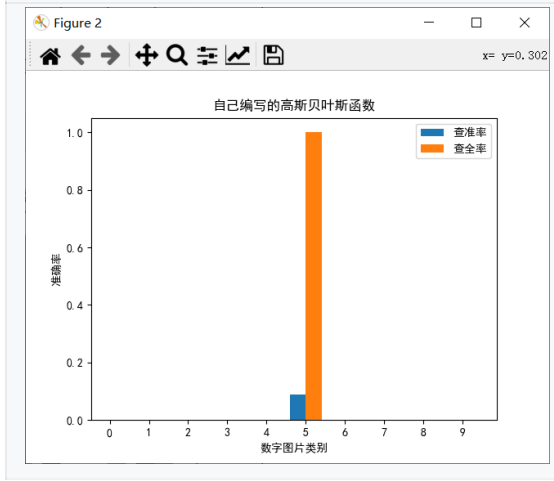
这里直接利用原始的特征数据进行分类，相当于直接使用离散属性。

```
************
自己编写的函数的准确率
[7 2 1 ... 4 5 6]
[5 5 5 ... 5 5 5]
892
0.0892
库函数的准确率
5558
0.5558
加载数据和处理耗时:0.32s
训练模型耗时:0.37s
预测耗时:2.02s
调库用的时间:1.28s
```

```
自己编写的函数在每个数字上的准确率
查准率：
[   nan    nan    nan    nan    nan 0.0892    nan    nan    nan    nan]
查全率：
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
高斯库函数在每个数字上的准确率
查准率：
[0.79019074 0.84561129 0.9047619  0.70883534 0.88421053 0.55
 0.64996369 0.87774295 0.28421053 0.36943907]
查全率：
[0.8877551  0.95066079 0.25775194 0.34950495 0.17107943 0.04932735
 0.934238   0.27237354 0.66529774 0.94648167]
多项式库函数在每个数字上的准确率
查准率：
[0.92494929 0.91229579 0.89842932 0.79756326 0.84331797 0.86005831
 0.88568486 0.93586957 0.66353544 0.71145919]
查全率：
[0.93061224 0.93480176 0.83139535 0.84257426 0.74541752 0.66143498
 0.89770355 0.83754864 0.79774127 0.85530228]
```







结果分析：结合结果可以看出，机器学习库中提供的高斯贝叶斯和多项式分类器都可以进行分类。虽然结果并不好。

但是自己编写的高斯贝叶斯分类器就无法处理这种连续值，将所有的测试样本都分为了数字5。

原因是图片数据是离散值，并且都是像素构成，每一类中的一个像素点的分布大概率不满足正态分布的。

数字分类准确率的分析

高斯贝叶斯在数字4和5的查全率明显低于其他数字，而数字8和9的查准率又明显低于其他数字，原因可能是

数字4，5的像素分布和8，9分布很类似，将大量的4，5数字错误的分类到了8，9上。

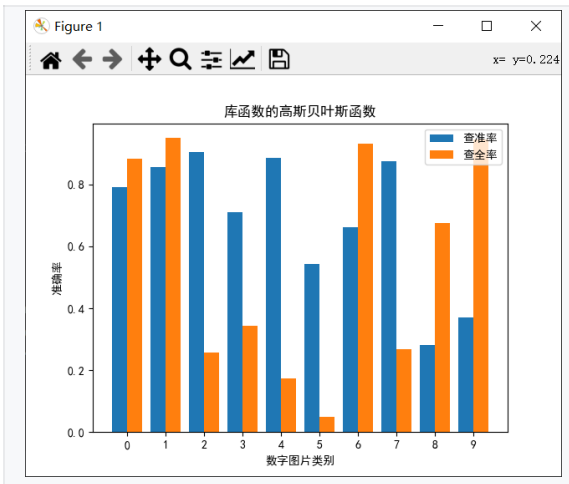
而数字0，1的查准率和查全率都很高是因为数字0，1的像素分布和其他都有较明显的区别，它们的特征更加的明显。

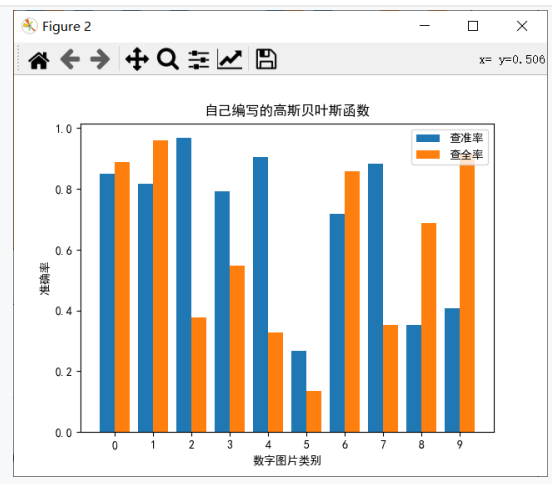
（2）原始数据不进行处理得到的模型分类准确率太低。这里对离散特征进行归一化处理，

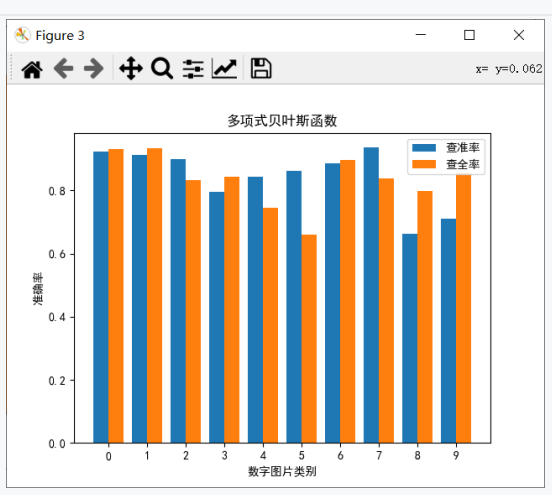
将离散特征进行连续化。

```
自己编写的函数的准确率
[7 2 1 ... 4 5 6]
[7 5 1 ... 9 8 6]
6131
0.6131
高斯库函数的准确率
5546
0.5546
多项式库函数的准确率
0.8364
加载数据和处理耗时:0.76s
训练模型耗时:0.34s
预测耗时:1.94s
调库用的时间:1.33s
```

```
自己编写的函数在每个数字上的准确率
查准率:
[0.84907498 0.81722846 0.96774194 0.79142857 0.9047619  0.26829268
 0.71752398 0.88264059 0.35297212 0.40748899]
查全率:
[0.88979592 0.96123348 0.37790698 0.54851485 0.32892057 0.13565022
 0.85908142 0.35116732 0.6889117  0.91674926]
高斯库函数在每个数字上的准确率
查准率:
[0.79120879 0.85487708 0.9047619  0.70901639 0.88541667 0.54320988
 0.66270431 0.87539936 0.2801192  0.36919505]
查全率:
[0.88163265 0.94977974 0.25775194 0.34257426 0.17311609 0.04932735
 0.93110647 0.26653696 0.67556468 0.94549058]
多项式库函数在每个数字上的准确率
查准率:
[0.92401216 0.91151203 0.89853556 0.79681648 0.84429066 0.86090776
 0.88648091 0.9359392  0.66296928 0.71145919]
查全率:
[0.93061224 0.93480176 0.83236434 0.84257426 0.74541752 0.65919283
 0.89665971 0.8385214  0.79774127 0.85530228]
```







结果分析：结合结果可以看出，多项式分类模型的准确率远高与其他两类。

原因应该是即使进行了归一化的处理，但是这并没有改变每个特征取值分布的情况，分布仍然很大概率不是正态分布，因此多项式的准确率很高。

自己训练的高斯模型准确率高于机器学习库的概率，很可能是在编写时，处理方差为0时的不同，自己训练的那个如果遇到反差为0的情况，无法计算时，会将方差赋予一个较小的值。

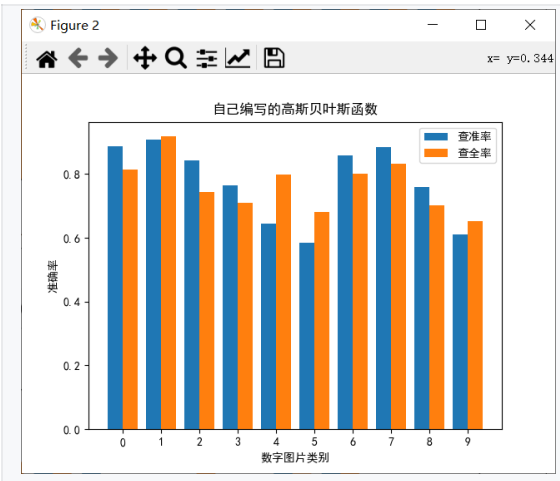
库函数的和归一化处理前的准确率几乎一样，说明库函数在进行训练的时候内部就已经进行过了归一化处理了。因此高斯库函数才能处理离散特征。

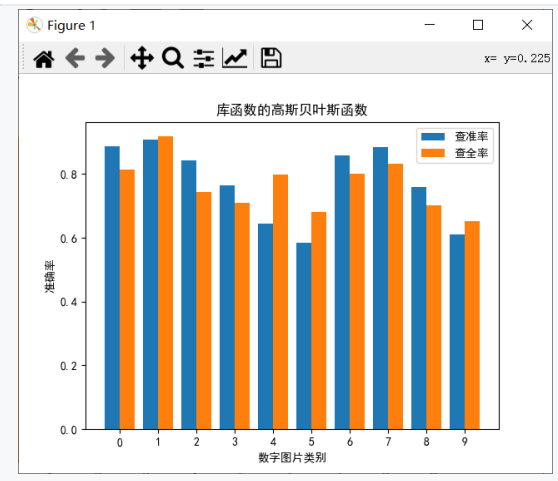
进行PCA降维后评价准确率。

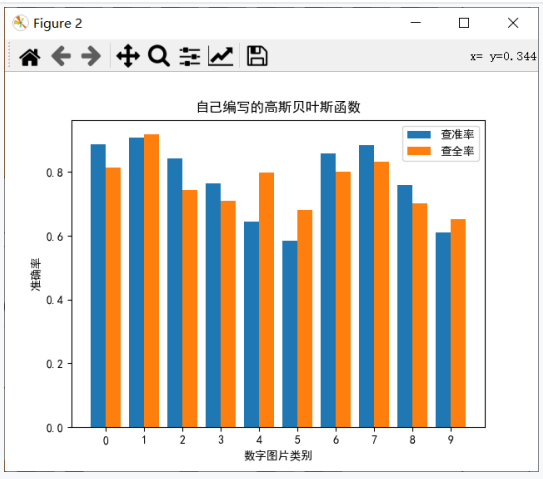
（1）PCA降维到10维度上

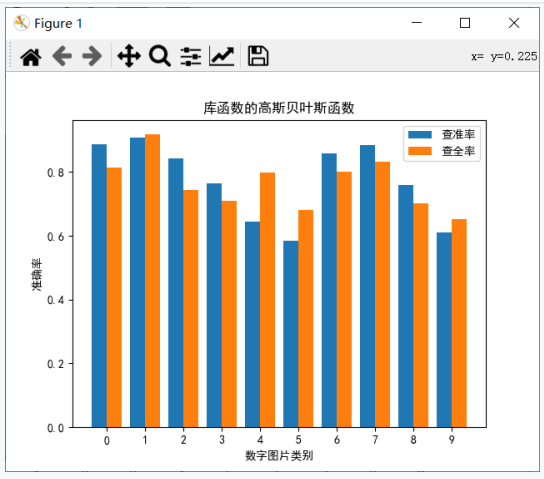
```
************
自己编写的函数的准确率
[7 2 1 ... 4 5 6]
[7 3 1 ... 9 5 6]
7678
0.7678
高斯库函数的准确率
7678
0.7678
多项式库函数的准确率
0.7158
加载数据和处理耗时:1.95s
训练模型耗时:0.02s
预测耗时:0.02s
调库用的时间:0.03s
```

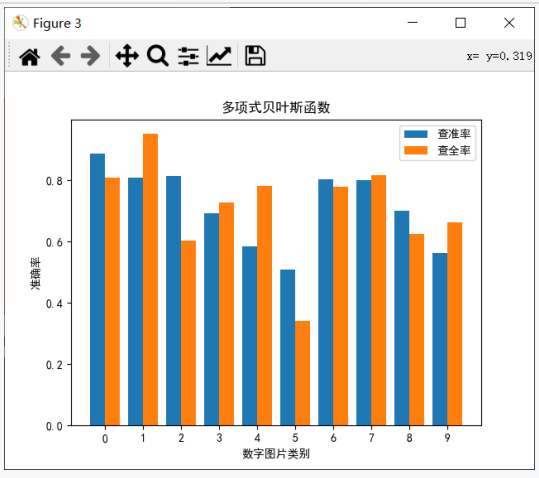
```
自己编写的函数在每个数字上的准确率
查准率:
[0.8865406  0.907585   0.84210526 0.76414088 0.64367816 0.58405379
 0.85794183 0.88314374 0.75747508 0.6090573 ]
查全率:
[0.81326531 0.91718062 0.74418605 0.70891089 0.79837067 0.68161435
 0.8006263  0.8307393  0.70225873 0.6531219 ]
库函数在每个数字上的准确率
查准率:
[0.8865406  0.907585   0.84210526 0.76414088 0.64367816 0.58405379
 0.85794183 0.88314374 0.75747508 0.6090573 ]
查全率:
[0.81326531 0.91718062 0.74418605 0.70891089 0.79837067 0.68161435
 0.8006263  0.8307393  0.70225873 0.6531219 ]
```











结果分析：降维后多项式的准确率不如高斯，这可能是因为，降维改变了数据的分布，并且这里是连续的情况，对于

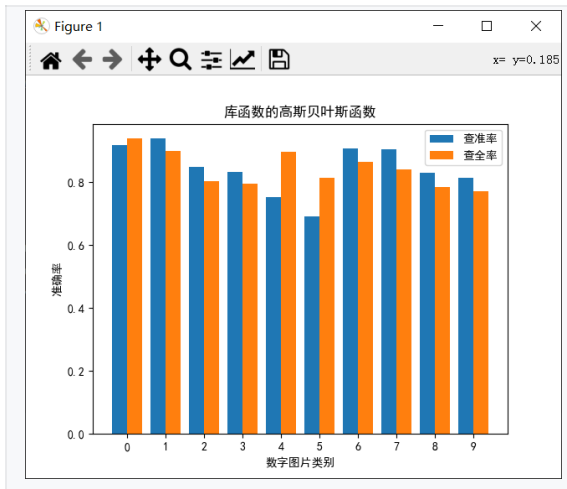
符合正态分布的连续值情况，多项式效果并不好。

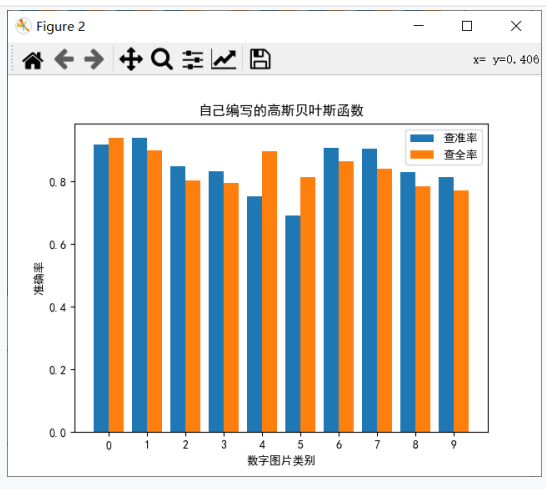
降维后数字的识别的准确率提高的很多，很大一部分提高在中间的数字部分，如3，4，5等数字的查准率和查全率有了明显的提高。

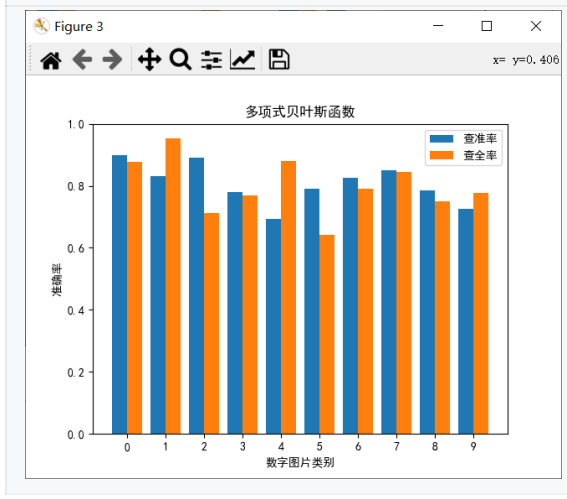
（2）PCA降维到20维度

```
************
自己编写的函数的准确率
[7 2 1 ... 4 5 6]
[7 5 1 ... 9 5 6]
8418
0.8418
高斯库函数的准确率
8418
0.8418
多项式库函数的准确率
0.8029
加载数据和处理耗时:2.19s
训练模型耗时:0.02s
预测耗时:0.05s
调库用的时间:0.06s
```

```
自己编写的函数在每个数字上的准确率
查准率：
[0.91625125 0.93767186 0.84994861 0.83367769 0.75192472 0.69333333
 0.90879121 0.90756303 0.82702703 0.81021898]
查全率：
[0.9377551  0.90132159 0.80135659 0.7990099  0.89511202 0.8161435
 0.86325678 0.84046693 0.78542094 0.77006938]
库函数在每个数字上的准确率
查准率：
[0.91625125 0.93767186 0.84994861 0.83367769 0.75192472 0.69333333
 0.90879121 0.90756303 0.82702703 0.81021898]
查全率：
[0.9377551  0.90132159 0.80135659 0.7990099  0.89511202 0.8161435
 0.86325678 0.84046693 0.78542094 0.77006938]
```





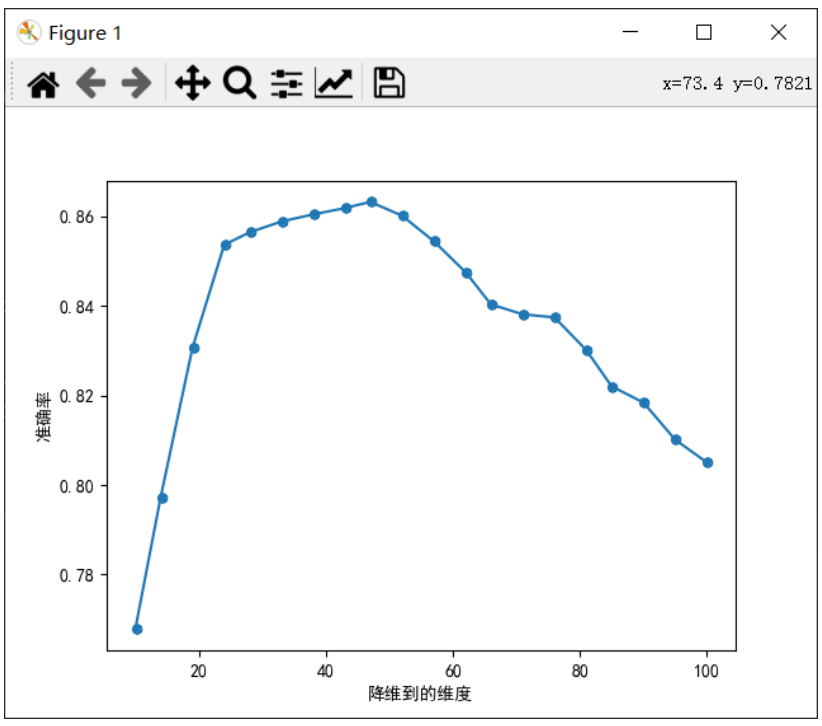


结果分析：分类的准确率都得到了提高。

降维到20维度的正确率高于10维的原因，可能是20维的特征保留了更多的原始数据的信息。

10维相对来说，损失了较多的信息。

（3）对高斯贝叶斯将数据降维到不同维度，进行准确率评估。



结果分析：维度过高或者过低都会降低准确率。

可能原因是，降维到太低维度可能会损失过多的信息。

降维太高，可能是有些特征的相关度可能较高。

维度在50维的时候准确率最高。降维需要选择合适的维度。

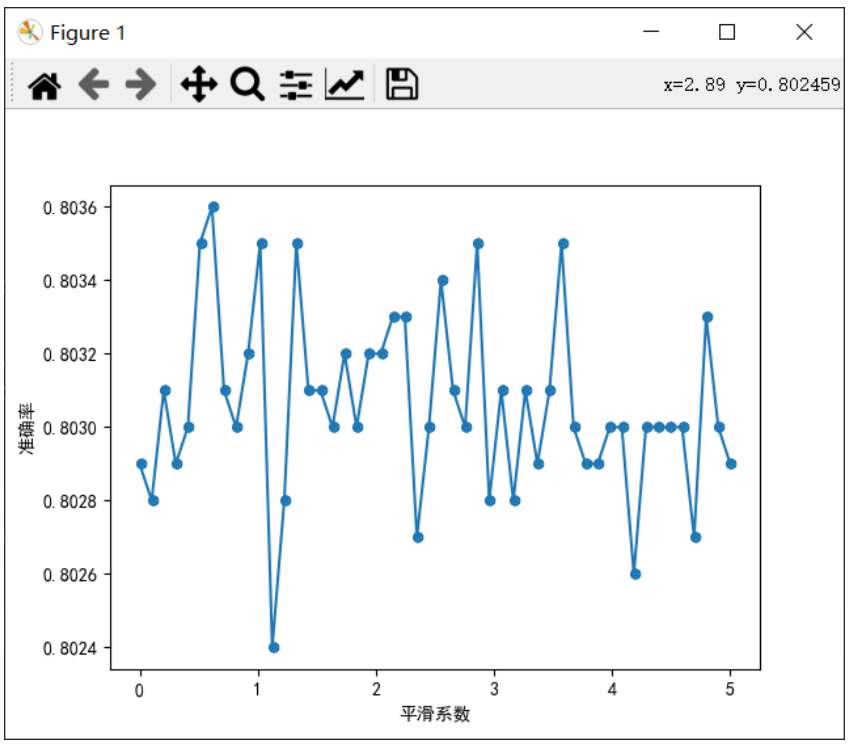
代码：

```
def minist_train_dio():

    dio = np.linspace(10,100,20,dtype=int)
    acc = np.zeros(len(dio))
    for i in range(len(dio)):
        print('数据开始运行')
        start_load_time = time.time()
        (x_train, y_train), (x_test, y_test) = mnist.load_data()

        x_train = imageMatrix_to_hang(x_train)
        x_test = imageMatrix_to_hang(x_test)

        # # 二值化
        # x_train = two_value(x_train,255/2)
        # x_test = two_value(x_test,255/2)

        # PCA降维
        print('进行pca降维')
        pca = PCA(n_components=dio[i])  # 加载PCA算法，设置降维后主成分数目为2
        pca.fit(x_train)
        x_train = pca.transform(x_train)  # 对样本进行降维
        x_test = pca.transform(x_test)
        #
        #
        # 归一化
        min_max_scaler = preprocessing.MinMaxScaler(feature_range=(0, 10))
        x_train = min_max_scaler.fit_transform(x_train)
        x_test = min_max_scaler.fit_transform(x_test)
        gaosi_x = sklearn.naive_bayes.GaussianNB()
        # gaosi_x = sklearn.naive_bayes.MultinomialNB()
        gaosi_x.fit(x_train, y_train)
        y_p_gaosi = gaosi_x.predict(x_test)
        end_ku_time = time.time()
        print('库函数的准确率')
        print(np.sum(y_test == y_p_gaosi))
        print(np.sum(y_test == y_p_gaosi) / len(y_p_gaosi))
        acc[i] = np.sum(y_test == y_p_gaosi) / len(y_p_gaosi)

    plt.figure()
    plt.rcParams["font.sans-serif"] = "SimHei"
    plt.rcParams["axes.unicode_minus"] = False
    plt.plot(dio,acc,marker='$\circledR$')
    plt.xlabel('降维到的维度')
    plt.ylabel('准确率')
    plt.show()

```

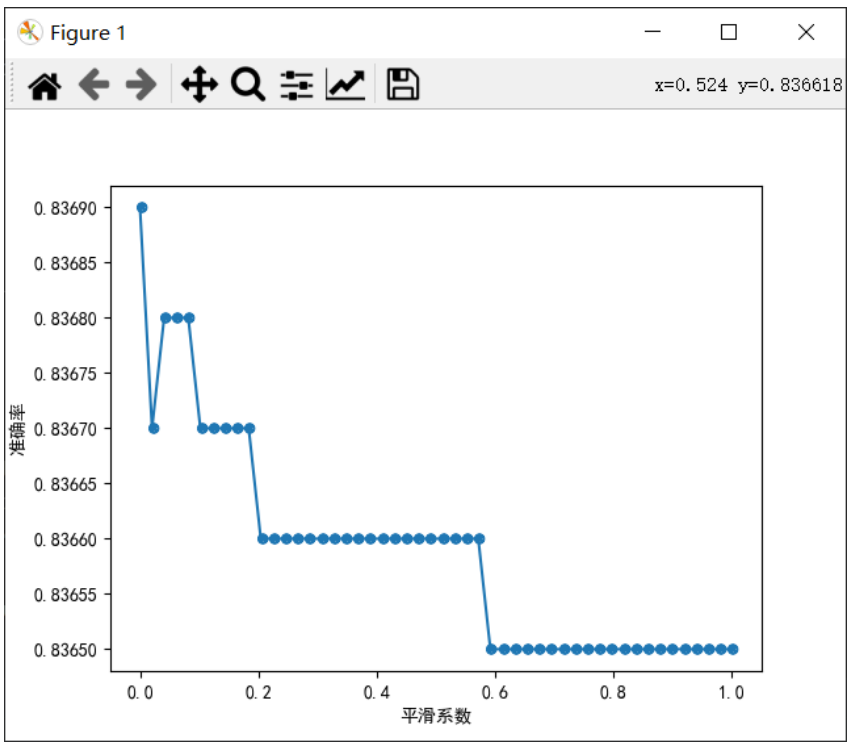
平滑系数对多项式贝叶斯分类器的影响。

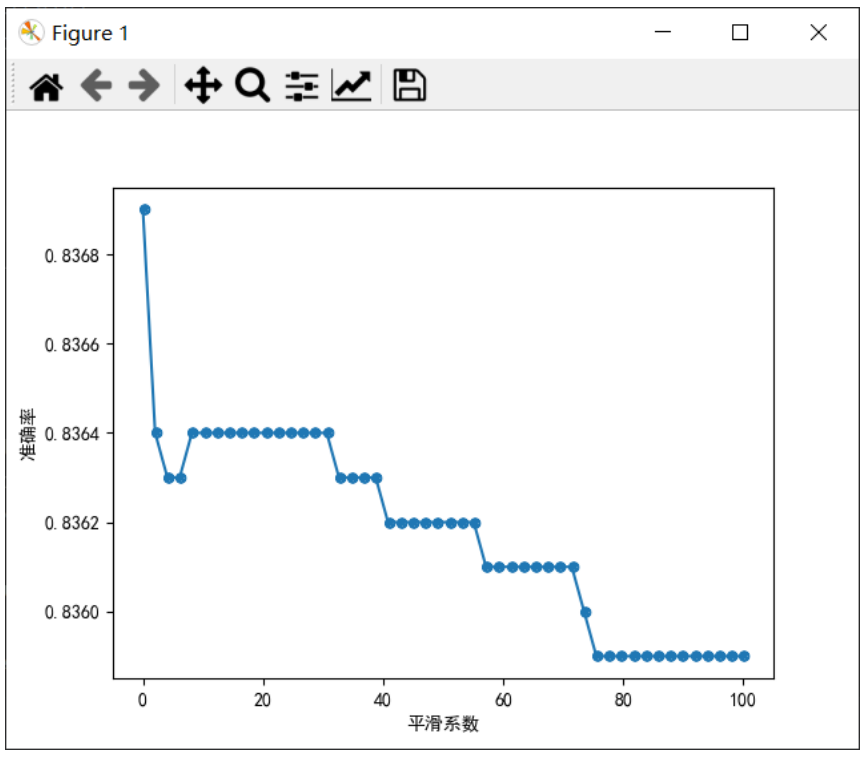
（1）数据在降维和归一化后的，平滑系数的影响比较杂乱，不能看出总体

是什么影响。



（2）数据不进行降维和归一化





结果分析：无论平滑系数在一个小的范围内，还是在一个大的范围内，准确率都会随着平滑系数的

增大而降低。

代码：

```
def minist_train_lapu():

    lapulasi = np.linspace(0,5,50)

    acc = np.zeros(len(lapulasi))
    for i in range(len(lapulasi)):
        print('数据开始运行')
        start_load_time = time.time()
        (x_train, y_train), (x_test, y_test) = mnist.load_data()

        x_train = imageMatrix_to_hang(x_train)
        x_test = imageMatrix_to_hang(x_test)

        # PCA降维
        print('进行pca降维')
        pca = PCA(n_components=20)  # 加载PCA算法，设置降维后主成分数目为2
        pca.fit(x_train)
        x_train = pca.transform(x_train)  # 对样本进行降维
        x_test = pca.transform(x_test)


        # 归一化
        min_max_scaler = preprocessing.MinMaxScaler(feature_range=(0, 10))
        x_train = min_max_scaler.fit_transform(x_train)
        x_test = min_max_scaler.fit_transform(x_test)
```

```
        gaosi_x = sklearn.naive_bayes.GaussianNB()
        gaosi_x = sklearn.naive_bayes.MultinomialNB(alpha=lapulasi[i])
        gaosi_x.fit(x_train, y_train)
        y_p_gaosi = gaosi_x.predict(x_test)
        end_ku_time = time.time()
        print('库函数的准确率')
        print(np.sum(y_test == y_p_gaosi))
        print(np.sum(y_test == y_p_gaosi) / len(y_p_gaosi))
        acc[i] = np.sum(y_test == y_p_gaosi) / len(y_p_gaosi)

    plt.figure()
    plt.rcParams["font.sans-serif"] = "SimHei"
    plt.rcParams["axes.unicode_minus"] = False
    plt.plot(lapulasi,acc,marker='$\circledR$')
    plt.xlabel('平滑系数')
    plt.ylabel('准确率')
    plt.show()
```

三、 总结（心得体会）

在这个实验中，我明白了在进行写代码这前最好进行数据分析，这可以减小写代码的时间，防止出现代码一直修改的情况。并且在开始写代码之前，最好要清楚你需要计算的数据，以及中间用到的数据的存储形式。

实验中不光要考虑代码是否能出结果，代码的运行效率同样重要，python中，能转换为矩阵计算的公式，最好化为矩阵计算，这样可以大幅提高代码的性能。

朴素贝叶斯在预测和训练速度相较于其他都比较块，但是他的准确率这些有上限，对于一般实际中的数据达到一定的准确度，就很难在贝叶斯的基础上提高太多。