**智能信息系统综合实践**

**实验报告**

|  |  |
| --- | --- |
| **题 目：** | **贝叶斯分类器 + PCA** |
| **年 级：** | **2021** |
| **专 业：** | **软件工程** |
| **学 号：** | **2021117405** |
| **姓 名：** | **孙潇桐** |

目录

[1. 题目 3](#_Toc162955213)

[2. 解题步骤 3](#_Toc162955214)

[2.1. 题目一：手动实现贝叶斯分类器对iris数据集进行分类 3](#_Toc162955215)

[2.1.1. 对数据的处理 3](#_Toc162955216)

[2.1.2. 构建贝叶斯分类器 3](#_Toc162955217)

[2.1.3. 测试使用所有特征组合作为分类依据的准确率 5](#_Toc162955218)

[2.1.4. 结果的展示和分析 6](#_Toc162955219)

[2.2. 题目二：使用朴素贝叶斯以及PCA对MNIST数据集分类 8](#_Toc162955220)

[2.2.1. 使用朴素贝叶斯对MNIST数据集分类 8](#_Toc162955221)

[2.2.2. 使用PCA降维 11](#_Toc162955222)

[2.2.3. 调节平滑系统参数 14](#_Toc162955223)

[3. 总结 16](#_Toc162955224)

[4. 附件 17](#_Toc162955225)

[4.1. 完整代码 17](#_Toc162955226)

[4.1.1. 不调库实现贝叶斯分类器分类iris数据集：iris.py 17](#_Toc162955227)

[4.1.2. 对MNIST数据集分类并展示各个数字的准确率：mnist.py 20](#_Toc162955228)

[4.1.3. 在4.1.2的基础加上降维操作：mnist\_with\_pca.py 22](#_Toc162955229)

[4.1.4. 在4.1.3的基础加上平滑操作：mnist\_with\_smoothing.py 26](#_Toc162955230)

[4.1.5. 工具类：utils.py 28](#_Toc162955231)

# 题目

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。尽量手动编写朴素贝叶斯分类代码，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。
2. 用MNIST数据集，随机选择70%训练,30%测试。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。利用PCA将28\*28的维度降维到某个维度（如10，20等），完成分类并计算其准确率。贝叶斯中唯一的参数一平滑系统，可尝试调整参数，比较其对分类的影响。

# 解题步骤

## 题目一：手动实现贝叶斯分类器对iris数据集进行分类

### 对数据的处理

通过观察，Iris数据集里面有三类样本 [‘Setosa’, ‘Versicolor’, ‘Virginica’]

接下来需要将得到的数据导入内存，从而完成接下来的计算。因为我在后面选择的**交叉验证**方式是**留出法**，所以根据以往的经验，我将测试集设置为样本总数的20%。设置随机种子为42以保证每次得到的数据集都是一样的，方便结果之间的比较，下面是我的实现代码，其中的load\_data函数是我写的一个写在utils.py文件中的工具函数，用于从csv文件中读取数据，完整的代码我会在文档的最后贴出。

# 加载数据

X, Y, name\_attr = utils.load\_data("data/iris.csv")

# 划分数据

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, Y, test\_size=0.2, random\_state=42

)

经过这样的处理，就得到了训练集的样本集和标签以及测试集的样本和标签。

### 构建贝叶斯分类器

在上个学期我曾经实现过离散数据的贝叶斯分类器，但是这次的实验样本集全是连续的特征值，所以我这次实现了一个能够分类连续特征值的朴素贝叶斯分类器。

贝叶斯分类器的原理是通过贝叶斯公式计算在某个特征值下的样本属于每个类别的概率，再从这些概率中选择最大的作为最后的预测。实现这样效果的关键在于我们需要使用数据集中已知的 来得到在不同特征值上的，而完成这个工作的就是贝叶斯分类器名字的来源，贝叶斯公式：

对我们的问题特化之后可以写成：

经过上面的公式我们就能通过计算样本中所有的从而推测出拥有某个特征的样本的类别。但是此处的特征是一个整体，也就是一个样本所有特征的集合。很显然我们在预测的时候难以得到和样本完全一样的特征，所以我们可以假定这些特征之间**相互独立**，这样的话我们就能通过将每个特征的概率分开计算，形式化的讲就是：

特征之间相互独立就是朴素贝叶斯分类器的重要假设，这个假设使得我们只需要计算每个类别下所有特征的可能性就可以完成对样本的预测。而且，对于同一个样本来说 在每个类别的可能性计算中都是固定的，所以在计算的时候可以忽略贝叶斯公式中的分母，也就是通过下面的式子得到可能性最大的类别：

对于离散数据的很好计算，只需要计算每个种类中的特征频率就行。但是本次实验我们需要预测连续值样本，此时就需要概率密度函数。我们可以假定这个特征值符合正态分布，也就是然后就可以通过计算每个种类的所有特征值的方差和平均值使用概率密度函数计算，也就是：

得到了方法，接下来就是我对上面算法的具体实现。我们首先需要计算所有样本下的所有特征值的平均值和方差，在这里我使用了numpy库中非常方便的布尔下标。通过计算布尔下标我们就可以很方便的从数据中取出我们想要的条目，无需对数据进行排序。数组p 记录的是每个种类的频率，二维数组mean和var分别记录的是每个种类下，所有特征值的平均值和方差。提前计算这些值可以在接下来的计算中加快运算速度，因为他们都是被重复使用的。下面是我对具体实现：

# 根据标签划分数据集并计算每个种类下的平均值和方差

X\_train\_divided = [[] for \_ in name\_tags]

# 每个种类的频率

p = [0.0] \* num\_tag

# mean 和 var 第一个下标代表种类，第二个下标代表特征

mean = X\_train\_divided.copy()

var = X\_train\_divided.copy()

for idx, tag in enumerate(name\_tags):

    # 得到当前种类的布尔索引

    mask = y\_train == tag

    X\_train\_divided[idx] = X\_train[mask]

    # 计算当前种类所有特征值的均值和方差和频率

    mean[idx] = np.mean(X\_train\_divided[idx], axis=0)

    var[idx] = np.mean(X\_train\_divided[idx], axis=0)

    p[idx] = len(X\_train\_divided[idx]) / num\_train\_sample

### 测试使用所有特征组合作为分类依据的准确率

想要测试所有的特征组合，可以很轻易的想到使用二进制枚举完成这个工作。所谓二进制枚举就是使用某一个数的每个二进制位来表示是否选择某一个特征作为判断依据，例如数字5可以写成，此时其二进制位就可以作为掩码来指示我们使用了第一个和第三个特征，没有使用第二个特征。

在这个数据集中有四个特征值，所以我们需要四位的掩码。也就是从到，而且根据二进制数的性质，从0加到15的过程中会经过所有的掩码组合。于是通过解析掩码得到布尔数组之后就可以很方便的选择我们需要的特征值，我也写了个函数用于将掩码转化为布尔数组：

def mask\_conv(mask):

    idx = [False] \* num\_attr

    for i in range(num\_attr):

        if mask & (1 << i) > 0:

            # 将当前位上为1的下标置1

            idx[i] = True

    idx.reverse()

    return idx

有了上面的工作，接下来只需要将从1到 之间的数都作为掩码使用上面提到的公式(4)和公式(5)计算出结果就行，下面是我的具体实现：

# 对每种特征组合进行测试

for mask in range(1, 1 << num\_attr):

    # 根据掩码得到布尔索引

    idx = mask\_conv(mask)

    # 选出需要的特征值

    x = X\_test[:, idx]

    # 记录结果

    res = np.zeros((x.shape[0], num\_tag))

    # 计算每一个种类的可能性

    for cur\_type in range(num\_tag):

        frac = 1 / (np.sqrt(2 \* np.pi) \* np.sqrt(var[cur\_type][idx]))

        res[:, cur\_type] = (

            np.prod(

                frac

                \* np.exp(-((x - mean[cur\_type][idx]) \*\* 2) / (2 \* var[cur\_type][idx])),

                axis=1,

            )

            \* p[cur\_type]

        )

    # 选择可能性最高的类型

    y\_predict = [name\_tags[i] for i in np.argmax(res, axis=1)]

    # 计算正确率

    accuracy = np.sum(y\_predict == y\_test) / y\_test.shape[0]

    # 存储结果

    arr\_attrs.append(name\_attr[idx])

    arr\_acc.append(accuracy)

    # print(f"{name\_attr[idx]}, {accuracy}")

我在上面的代码为了效率，大量的使用了numpy的广播机制，其调用了numpy的底层c代码库，避免了在python中使用循环，不仅提高了计算效率（经过我对测试有50倍的性能提升），还简化了代码，使代码更接近数学公式。

### 结果的展示和分析

为了更好的展示使用不同特征值组合下的准确率，我使用了柱状图来直观的展示结果。下面是我对画图过程：

# 展示柱状图

plt.figure(figsize=(14, 7))

bars = plt.bar(

    np.arange(len(arr\_acc)),

    arr\_acc,

    tick\_label=arr\_attrs,

)

# 在每个柱子上方添加具体数字

for acc, bar in zip(arr\_acc, bars):

    h = bar.get\_height() + 0.01

    x, y = bar.get\_x(), bar.get\_width()

    plt.text(x + y / 2, h, f"{acc:.3}", ha="center", va="bottom")

plt.xlabel("特征值组合")

plt.ylabel("准确率")

plt.title("所有特征值组合的准确率")

plt.xticks(rotation=20, ha="right")

plt.tight\_layout()

plt.show()

下面是我得到的结果：

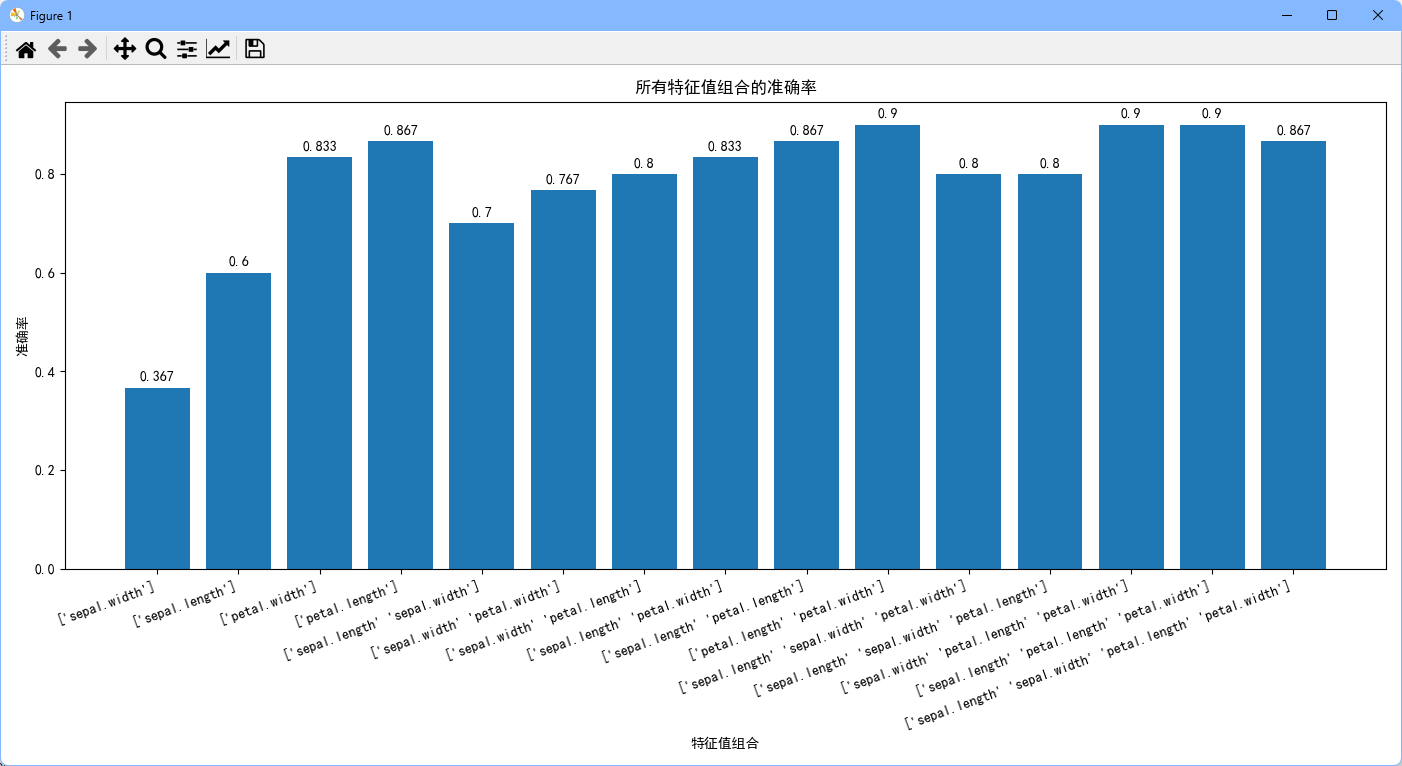


Figure 1 所有特征值组合的准确率

从上面的图可以很轻易的看出来，并不是特征值选择的越多越好，比如只用**花瓣宽度**分类就可以达到86.7%的准确率，和使用全部特征值分类的准确率一样。最高的准确率(90%)，均出现在同时使用**花瓣长度**和**花瓣宽度**这两个特征值的情况下。仅使用这两个特征就可以达到(90%)的准确率，且搭配剩下来的两个特征值也不会有准确率的提升。说明这两个特征（花瓣长度和花瓣宽度）是这个三分类问题的关键，其他的特征都不太重要，考虑过多反而可能导致正确率下降。

## 题目二：使用朴素贝叶斯以及PCA对MNIST数据集分类

### 使用朴素贝叶斯对MNIST数据集分类

因为前面的实验我并没有实现拉普拉斯平滑，且我在前面以及将朴素贝叶斯分类器的原理解释的比较清楚了，所以后面的实验我将使用sklearn提供的库完成，不仅更加高效，也能精简代码从而让我们看到更关键的部分。

我为了使用方便，将MNIST数据集下载之后以csv格式储存在本地，这样就可以使用我写的load\_data函数加载数据到内存中。接下来就从所有数据中随机取出30%作为测试集，剩下的作为训练集。下面是我的实现

# 加载数据

X, y, \_ = utils.load\_data("data/mnist.csv", 784, 784)

# 划分数据集为训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, y, test\_size=0.3, random\_state=42

)

接下来就使用贝叶斯分类器对训练集训练之后对测试集进行测试就可以得到预测所有数字的准确度，我在这里得到的准确度是**55.095%**，下面是我对具体实现：

# 初始化朴素贝叶斯分类器

nb\_classifier = GaussianNB()

# 训练模型

nb\_classifier.fit(X\_train, y\_train)

# 预测测试集

y\_pred = nb\_classifier.predict(X\_test)

# 计算准确率

accuracy = accuracy\_score(y\_test, y\_pred)

print("总体准确度:", accuracy)

在得到整体准确度之后，我对这个准确度并不是很满意。为了看看是哪个数字“拖了后腿”，我对每个数字的准确率进行了可视化。下面是我的画图过程：

# 计算每个数字的准确率

tags = np.unique(y\_test)

accuracys = []

for digit in tags:

    idx = y\_test == digit

    accuracys.append(accuracy\_score(y\_test[idx], y\_pred[idx]))

# 画出准确率图表

plt.figure(figsize=(12, 6))

bars = plt.bar(tags, accuracys)

# 在每个柱子上方添加准确率

for acc, bar in zip(accuracys, bars):

    h = bar.get\_height() + 0.01

    x, y = bar.get\_x(), bar.get\_width()

    plt.text(x + y / 2, h, f"{acc:.3}", ha="center", va="bottom")

plt.xlabel("数字")

plt.ylabel("准确率")

plt.title("各个数字的准确率")

plt.xticks(tags)

# plt.grid(True)

plt.show()

经过计算之后得到的结果如下图所示：

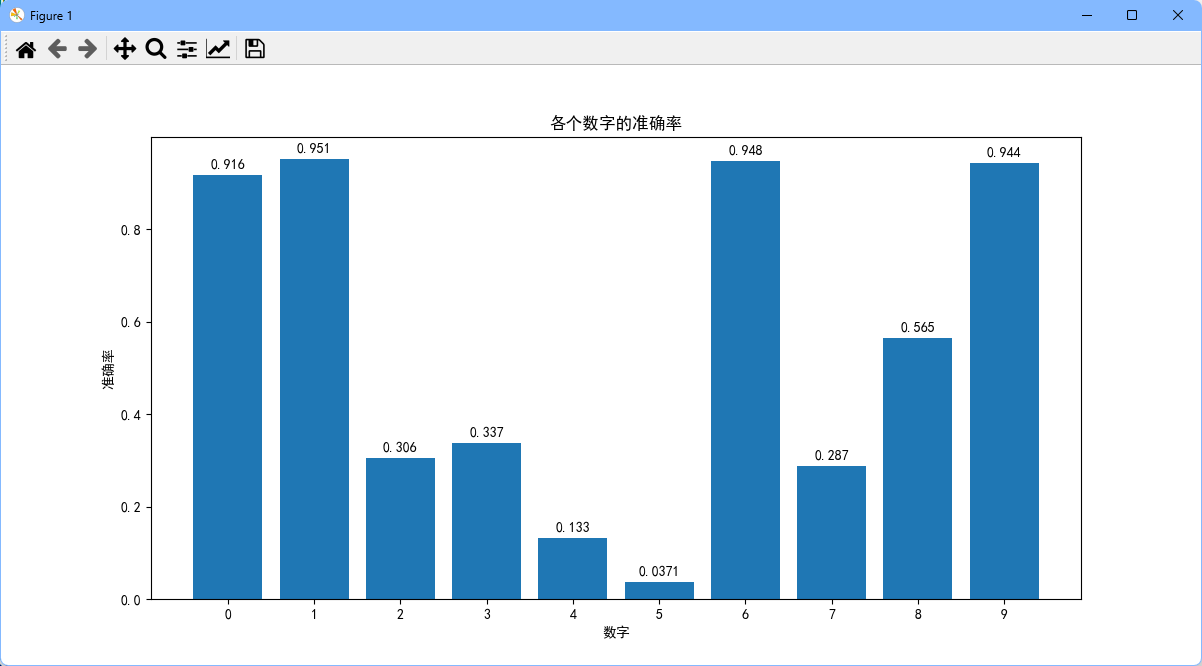


Figure 2 每个数字的准确率

可以看到对数字5的预测准确率最低，我在很好奇分类器把5预测成什么了的同时也想知道其他数字的预测情况，为了得到我想要的结果，我对这个分类器计算了其在训练集上的混淆矩阵以及每个种类的查准率，查全率和F1值。并使用了我在utils.py工具类中实现的展示函数优雅的展示了出来，完整的display函数代码会在结尾的附件中贴出。下面是我的实现：

# 多分类混淆矩阵，行为预测值，列为真实值

confusion\_matrix = np.zeros((tag\_num, tag\_num), dtype=int)

# 记录每个分类结果的真实值和预测值

for real, pred in zip(y\_test, y\_pred):

    confusion\_matrix[pred][real] += 1

# 查准率

precision = confusion\_matrix.diagonal() / np.sum(confusion\_matrix, axis=1)

# 查全率

recall = confusion\_matrix.diagonal() / np.sum(confusion\_matrix, axis=0)

# F1

f1 = 2 \* np.multiply(precision, recall) / (precision + recall)

# 展示结果（混淆矩阵，查准率，查全率和F1\_Score)

utils.display(confusion\_matrix, precision, recall, f1, tags)

下面是我得到的结果：

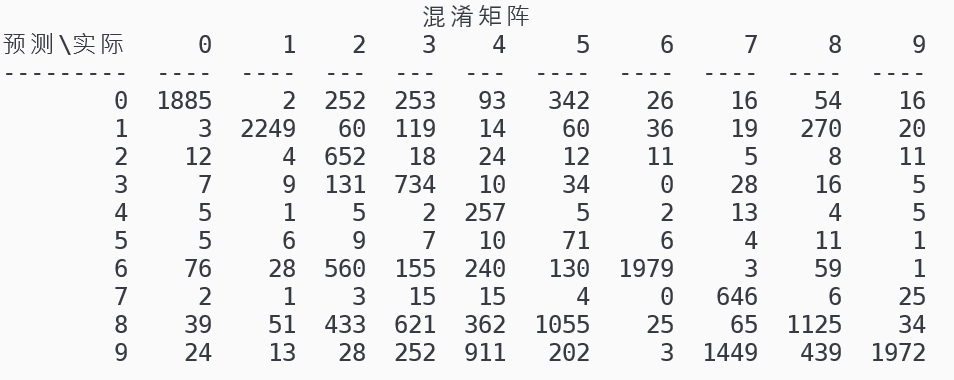


Figure 3 MNIST贝叶斯分类混淆矩阵

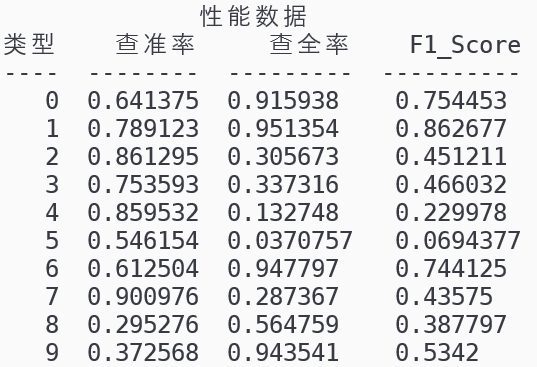


Figure 4 MNIST贝叶斯分类性能参数

通过观察混淆矩阵，可以很轻易的发现，大量的5被预测成了8。这应该是由于5的手写体与8太相似了，只要将5首尾相连就可以得到8。与5情况相似的还有数字4和7，从混淆矩阵可以看出有大量的4和7被分类成为了9，因为它们在手写的时候相似度比较高。

根据前面提到的贝叶斯分类器原理，其主要是考虑每一个像素在每种数字中出现的概率。因为没有像卷积神经网络那样提取整体特征的能力，很容易造成这种部分像素相似而分类错误的问题。

### 使用PCA降维

为了进一步改进我们的贝叶斯分类器，我们可以通过PCA（主成分分析）来对数据进行降维。而PCA分类后的数据集有两个非常重要的性质：

* 最大可分性：降维的本质是将高维的数据投影到低维，我们希望投影之后的值尽可能的**分散**，避免因样本重叠导致的信息损失。
* 最近重构性：在降维的过程中，尽量保证数据之间的距离不变，从而尽可能的保留数据的特征。

为了满足上面的两个性质，PCA使用了协方差矩阵。通过计算协方差矩阵C，我们就可以在保留数据方差的同时很方便的进行特征值分解得到特征值和特征向量。但是协方差之间如果特征具有不同的尺度，可能会导致主成分不准确，所以需要先进行**数据中心化**。为了得到特征值，我们需要对中心化之后的数据进行下面的计算：

先计算协方差矩阵，协方差代表当前点与中心点的偏差，可以以矩阵方式计算：

然后再对协方差矩阵C进行特征值分解，也就是求解来得到特征值之后再计算来计算具体的特征值。再取特征值中最大的 个特征值（ 指我们希望降维到的维度）对应的特征向量组成投影矩阵W（这些特征向量也叫做**主成分**），就可以使用投影矩阵与原始数据相乘得到降维之后的数据：

其中Y是降维之后的数据，X是中心化之后的数据，W是投影矩阵（其中的每个主成分纵向排列）。通过上述的一通计算，我们终于得到了降维之后的数据Y，为了探究降维之后的维度对贝叶斯分类器的影响，我将设置多个维度从5到200每次增加10，也就是：5, 15, 25… 共20个点。我将使用每次降维之后的数据训练贝叶斯分类器，从而比较不同维度之间的准确率。同时为了更加贴近显示应用，我没有让测试集里面的数据参与主成分分析，而是仅对训练集进行主成分分析，然后使用训练集主成分分析得到的投影矩阵W与测试集相乘达到降维的目的，接下来是我的具体实现：

# 测试降维的维度(5, 15, 25...)

dimensions = range(5, 200, 10)

# 记录准确率

accuracys = []

# 记录最高的准确率以及使用的参数（准确率，目标维度）

best\_prama = (-1.0, -1)

for i in utils.tqdm(dimensions):

    # 指定降维的维度

    pca = PCA(n\_components=i)

    # 计算投影矩阵W

    pca.fit(X\_train)

    W = pca.components\_.T

    # 通过投影矩阵将当前训练集降维

    X\_train\_pca = X\_train @ W

    # 初始化朴素贝叶斯分类器

    naive\_bayesian = GaussianNB()

    # 训练模型

    naive\_bayesian.fit(X\_train\_pca, y\_train)

    # 对测试数据集降维

    X\_test\_pca = X\_test @ W

    # 预测测试集

    y\_pred = naive\_bayesian.predict(X\_test\_pca)

    # 计算准确率

    accuracy = accuracy\_score(y\_test, y\_pred)

    if accuracy > best\_prama[0]:

        best\_prama = (accuracy, i)

    accuracys.append(accuracy)

    # print(f"降到 {i} 维时，准确率为 {accuracy:.4}")

print(f"最佳维度为{best\_prama[1]}维，准确率为{best\_prama[0]}")

在得到一系列数据之后，我对每个维度对应的准确率画出了折线图：

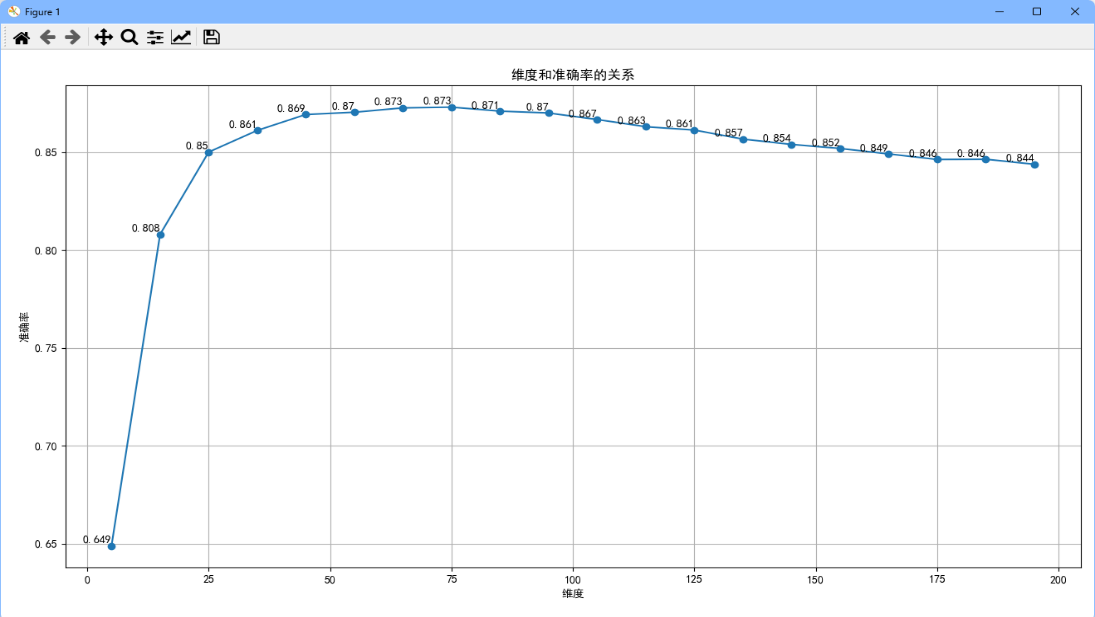


Figure 5 PCA降维的维度与准确率的关系

从上面的图表可以很轻易的看出来，哪怕是降维到5维，整体的准确率也达到了**64.9%**，已经超过我们在上文2.2.1节中使用完整784维数据得到的**55.095%** 准确率。这也就可以看出来PCA的一个非常重要的作用：降噪。经过测试我们的分类器得以在降维到75维的时候，达到**最高**为87.3%的准确率。

当然，我也很好奇之前在2.2.1节中分类器表现不佳的数字5,4和7在最佳分类器的准确度。于是我计算了降维到75维之后分类器的混淆矩阵和每个数字的准确率，代码和2.2.1节中的代码非常相似，这里就不贴出来了，完整代码可以在附录（4.1节）中查看，下面是我得到的结果：

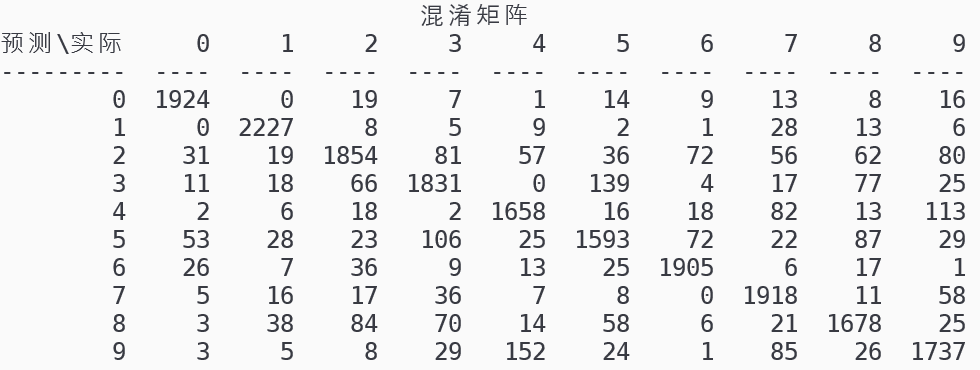


Figure 6 降维到75维后的混淆矩阵



Figure 7 降维到75维之后的性能数据

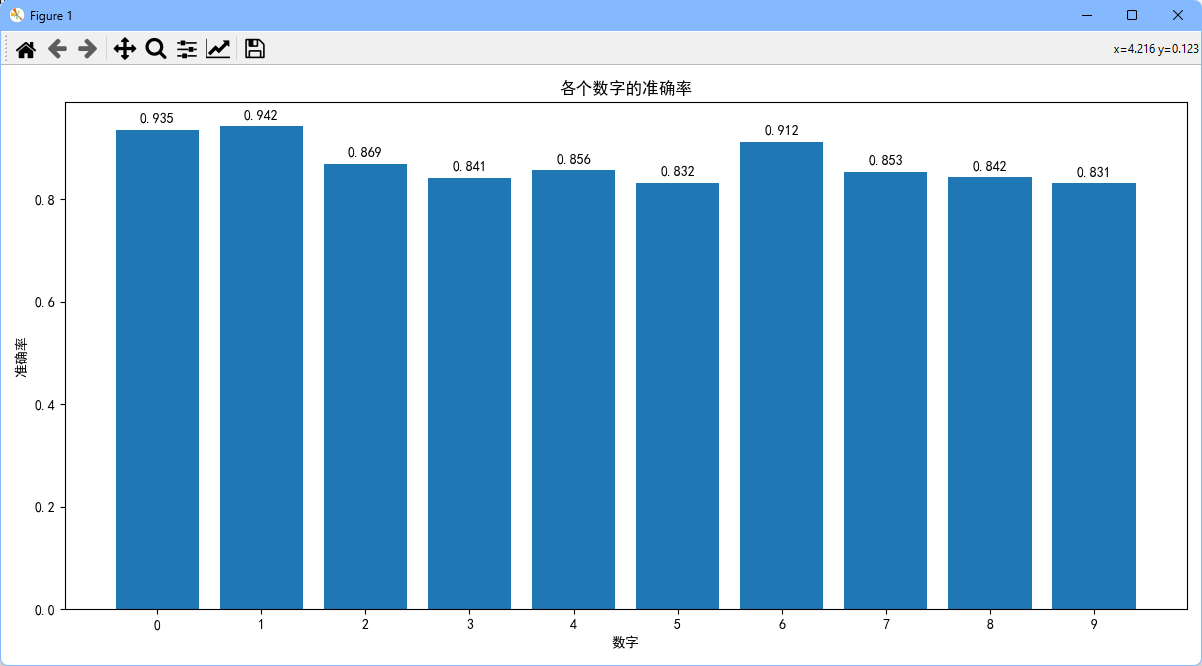


Figure 8 降维到75维的各个数字准确率

从上面的结果可以看出来，之前表现不佳的数字现在的准确率已经大幅提升，我认为这归功于降维的另一个性质：特征独立。在降维的过程中，数据的特征相互独立，使得分类器更加难以混淆类似于5和8这样相似的数字。通过降维，我们已经成功的使整体准确率从**55.095%** 大幅提高到 **87.3% 。**

### 调节平滑系统参数

在朴素贝叶斯分类器中，计算类别条件概率时，如果某个特征值在给定训练集中从未出现过，那么直接使用频数进行估计会导致预测样本的时候**概率为零**，从而影响到后续的分类结果。平滑参数的引入可以避免这种情况发生，通过在计算中添加一个小的偏置，使得即使某个特征值没有在训练集中出现，其概率也不会为零。

在离散的特征值中很好理解，但是对于连续值的拉普拉斯平滑，我们就需要对每个类别下的每个特征值的平均值和方差进行平滑。在预测的时候，对均值和方差具体操作可以通过下面两个公式完成：

其中I为是指示函数，当括号内条件成立时为1，否则为0。是平滑参数，接下来我们调节的都是公式中的。

在sklearn库中的朴素贝叶斯分类器有一个参数叫var\_smoothing，这个参数对应的就是我们上面说的平滑系数，我通过调节的值来探究平滑系数对分类结果的影响。我先在没有降维的数据上尝试，下面是我的实现过程：

smoothings = np.linspace(0.001, 0.5, 33)

accuracys = []

for i in utils.tqdm(smoothings):

    # 初始化朴素贝叶斯分类器

    naive\_bayesian = GaussianNB(var\_smoothing=i)

    # 训练模型

    naive\_bayesian.fit(X\_train, y\_train)

    # 预测测试集

    y\_pred = naive\_bayesian.predict(X\_test)

    # 计算测试集准确率

    accuracy = accuracy\_score(y\_test, y\_pred)

    accuracys.append(accuracy)

    # print(f"alpha为{i}的时候，测试集上的准确度{accuracy}")

idx = np.argmax(accuracys)

print(f"当aplha={smoothings[idx]:.3}时，最佳准确度为{accuracys[idx]:.3}")

我将平滑参数与准确率的曲线也画了出来：

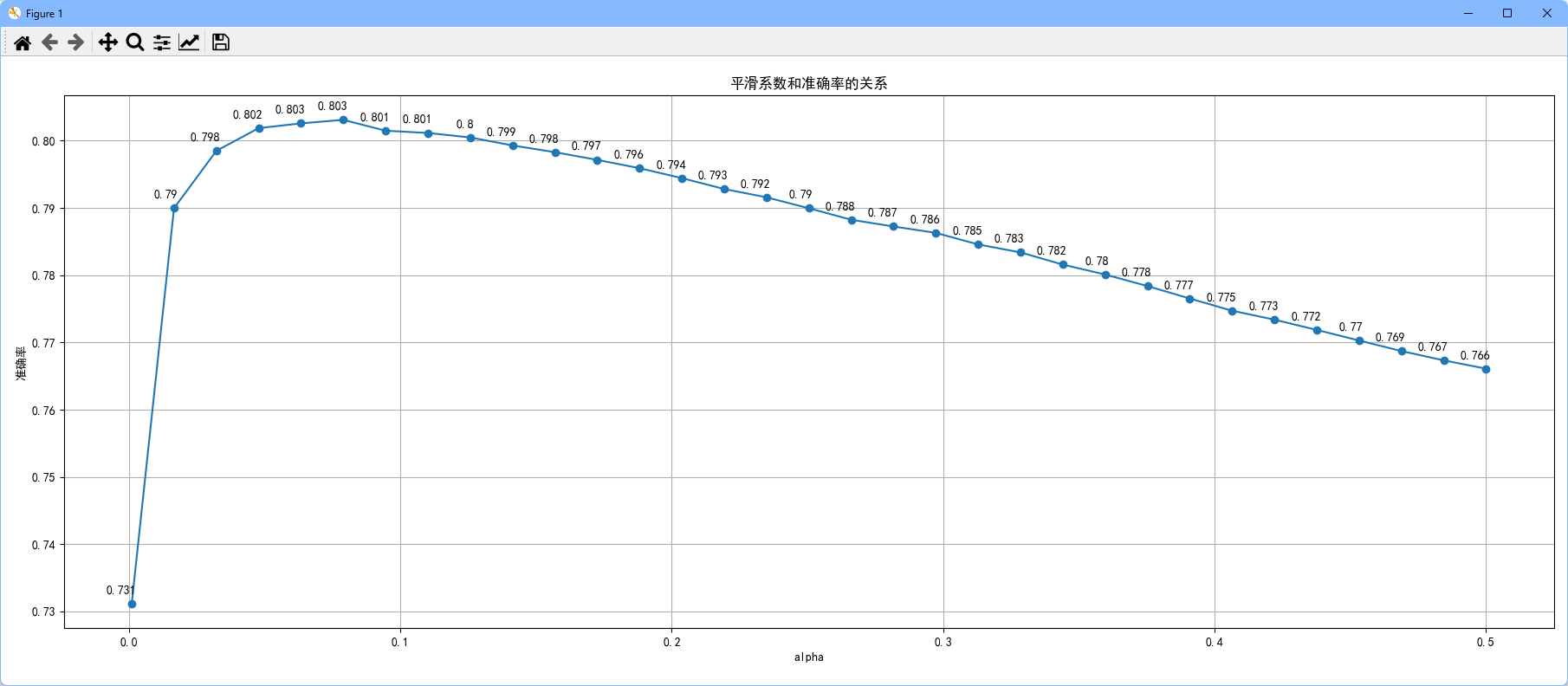


Figure 9 平滑系数和准确率的关系

从曲线中可以很轻易的看出，当平滑系数时，准确率达到最高的80.3%，这是还是没有降维的情况下。准确率就从**55.095%** 提升至 **80.3%**，可见拉普拉斯平滑对于这个数据集十分重要。我认为应该是手写数据集背景是黑色的，就导致784个特征值中有很多特征值为0，这些0严重的影响到了对测试集的分类，平滑后问题就能得到缓解。

既然在没有降维的情况下使用平滑的提升就已经如此之大，接下来我将根据上文（2.2.2节）中测试得到的最佳降维维度75维，对数据进行降维，并探究降维后的平滑参数对准确率的影响。我在之间选取了40个点，画出来平滑参数和准确率的折线图。由于代码重复度比较高，我就直接贴结果了，完整代码在附件可以查看。

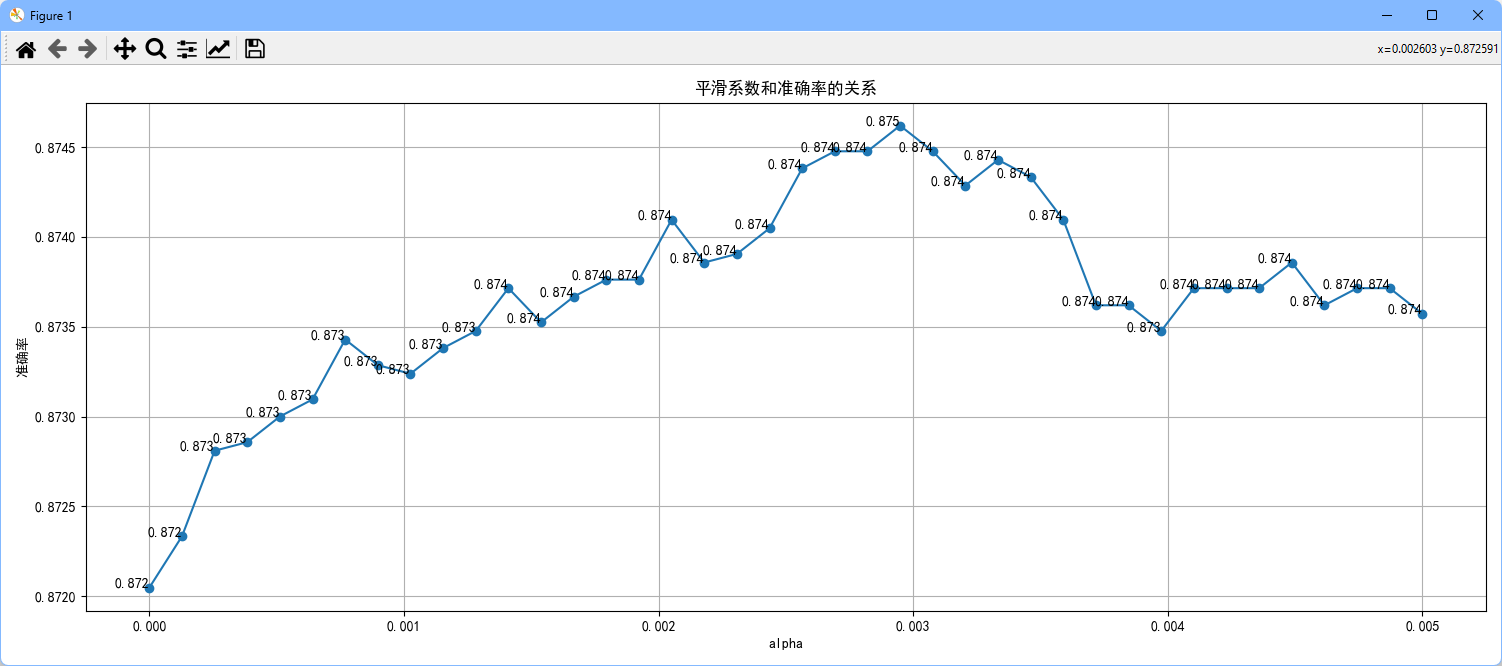


Figure 10 降维到75维之后平滑系数与准确率的关系

通过对图像的观察我们可以发现，当 时，最佳准确度为**87.5%**。可以参考我在上一节（2.2.2节）中不使用平滑得到的**87.3%**，设置平滑系数只让准确率最多上升**0.2%**，我认为是因为在降维的时候，少了很多特征值为0的位置，所以使用拉普拉斯平滑对准确率的提升不大。

# 总结

详细的结论我都直接写在了第二节中的每一次的结果后面，这里再对之前的结果进行一次概括。首先，从2.1.4.节对iris数据集的分类结果来看，贝叶斯分类器并不是使用越多特征越好，使用无关的特征值可能反而使准确率下降。其次，从2.2.2.节和2.2.3节中MNIST数据集的结果来看，贝叶斯分类器在预测大量从未出现过的值的时候，平滑非常重要，在这个数据集上平滑之后可以使正确率最高提升25% ，而且对于高维数据来说，PCA降维也很有必要，降维可以正确率最高提升32%，但是经过降维之后的数据，由于基本上不会出现预测时遇到大量未出现过的值，对所以降维后的数据再加平滑意义不大。

这次实验花费了我非常多的时间和精力，从复习上个学习学过的贝叶斯分类器的原理到可视化模型的方法，到思考模型的测试策略。每一个环节都十分重要，我也都遇到了一些困难。

这次我花费最多时间解决的问题就是在亲手实现朴素贝叶斯分类器的时候遇到的运行效率问题。一开始我使用普通的循环计算每个种类的可能性，iris数据集并不大，但是感觉速度非常慢。通过查阅资料，我了解到了numpy特有的广播机制，通过广播机制，可以将运算交给底层的高度优化的c代码库。于是我深入的学习了numpy，改进了计算方式，经过我的测试，大概收获了50倍的性能提升。

这次的实验让我受益匪浅，使我对numpy库和sklearn库的理解更进一步的同时也锻炼了我解决问题的能力。上面提到的两个库都是在机器学习领域非常重要的工具，只有用好他们才能更好的完成实验，并理解这些模型的底层原理。我将继续保持学习，多学，多写，争取有朝一日能成为机器学习大师。

# 附件

## 完整代码

### 不调库实现贝叶斯分类器分类iris数据集：iris.py

import utils

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from matplotlib import rcParams

# 设置中文字体

rcParams["font.family"] = "SimHei"

# 加载数据

X, Y, name\_attr = utils.load\_data("data/iris.csv")

# 划分数据

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, Y, test\_size=0.2, random\_state=42

)

# 得到特征值名称和数目

name\_attr = np.array(name\_attr)

num\_train\_sample, num\_attr = X\_train.shape

# 得到标签

name\_tags = np.unique(Y)

num\_tag = len(name\_tags)

# 根据标签划分数据集并计算每个种类下的平均值和方差

X\_train\_divided = [[] for \_ in name\_tags]

# 每个种类的频率

p = [0.0] \* num\_tag

# mean 和 var 第一个下标代表种类，第二个下标代表特征

mean = X\_train\_divided.copy()

var = X\_train\_divided.copy()

for idx, tag in enumerate(name\_tags):

    # 得到当前种类的布尔索引

    mask = y\_train == tag

    X\_train\_divided[idx] = X\_train[mask]

    # 计算当前种类所有特征值的均值和方差

    mean[idx] = np.mean(X\_train\_divided[idx], axis=0)

    var[idx] = np.mean(X\_train\_divided[idx], axis=0)

    # 计算当前种类的频率

    p[idx] = len(X\_train\_divided[idx]) / num\_train\_sample

# 将掩码转换为布尔索引

def mask\_conv(mask):

    idx = [False] \* num\_attr

    for i in range(num\_attr):

        if mask & (1 << i) > 0:

            # 将当前位上为1的下标置1

            idx[i] = True

    idx.reverse()

    return idx

# 储存使用的特征值和准确率

arr\_attrs = []

arr\_acc = []

# 对每种特征组合进行测试

for mask in range(1, 1 << num\_attr):

    idx = mask\_conv(mask)

    # 选出需要的特征值

    x = X\_test[:, idx]

    # 记录结果

    res = np.zeros((x.shape[0], num\_tag))

    # 计算每一个种类的可能性

    for cur\_type in range(num\_tag):

        frac = 1 / (np.sqrt(2 \* np.pi) \* np.sqrt(var[cur\_type][idx]))

        res[:, cur\_type] = (

            np.prod(

                frac

                \* np.exp(-((x - mean[cur\_type][idx]) \*\* 2) / (2 \* var[cur\_type][idx])),

                axis=1,

            )

            \* p[cur\_type]

        )

    # 选择可能性最高的类型

    y\_predict = [name\_tags[i] for i in np.argmax(res, axis=1)]

    # 计算正确率

    accuracy = np.sum(y\_predict == y\_test) / y\_test.shape[0]

    # 存储结果

    arr\_attrs.append(name\_attr[idx])

    arr\_acc.append(accuracy)

    # print(f"{name\_attr[idx]}, {accuracy}")

# 排序

sorted\_indices = np.lexsort((arr\_acc, [len(attr) for attr in arr\_attrs]))

arr\_attrs = [str(arr\_attrs[idx]) for idx in sorted\_indices]

arr\_acc = np.array(arr\_acc)[sorted\_indices]

# 展示柱状图

plt.figure(figsize=(14, 7))

bars = plt.bar(

    np.arange(len(arr\_acc)),

    arr\_acc,

    tick\_label=arr\_attrs,

)

# 在每个柱子上方添加具体数字

for acc, bar in zip(arr\_acc, bars):

    h = bar.get\_height() + 0.01

    x, y = bar.get\_x(), bar.get\_width()

    plt.text(x + y / 2, h, f"{acc:.3}", ha="center", va="bottom")

plt.xlabel("特征值组合")

plt.ylabel("准确率")

plt.title("所有特征值组合的准确率")

plt.xticks(rotation=20, ha="right")

plt.tight\_layout()

plt.show()

### 对MNIST数据集分类并展示各个数字的准确率：mnist.py

import numpy as np

import matplotlib.pyplot as plt

import utils

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score

from matplotlib import rcParams

# 设置中文字体

rcParams["font.family"] = "SimHei"

# 加载数据

X, y, \_ = utils.load\_data("data/mnist.csv", 784, 784)

y = np.array(y, dtype=int)

# 划分数据集为训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, y, test\_size=0.3, random\_state=42

)

# 初始化朴素贝叶斯分类器

naive\_bayesian = GaussianNB()

# 训练模型

naive\_bayesian.fit(X\_train, y\_train)

# 预测测试集

y\_pred = naive\_bayesian.predict(X\_test)

# 计算测试集准确率

accuracy = accuracy\_score(y\_test, y\_pred)

print("总体在测试集上的准确度:", accuracy)

# 计算每个数字的准确率

tags = np.unique(y\_test)

accuracys = []

for digit in tags:

    idx = y\_test == digit

    accuracys.append(accuracy\_score(y\_test[idx], y\_pred[idx]))

tag\_num = len(tags)

# 多分类混淆矩阵，行为预测值，列为真实值

confusion\_matrix = np.zeros((tag\_num, tag\_num), dtype=int)

# 记录每个分类结果的真实值和预测值

for real, pred in zip(y\_test, y\_pred):

    confusion\_matrix[pred][real] += 1

# 查准率

precision = confusion\_matrix.diagonal() / np.sum(confusion\_matrix, axis=1)

# 查全率

recall = confusion\_matrix.diagonal() / np.sum(confusion\_matrix, axis=0)

# F1

f1 = 2 \* np.multiply(precision, recall) / (precision + recall)

# 展示结果（混淆矩阵，查准率，查全率和F1\_Score)

utils.display(confusion\_matrix, precision, recall, f1, tags)

# 画出准确率图表

plt.figure(figsize=(12, 6))

bars = plt.bar(tags, accuracys)

# 在每个柱子上方添加准确率

for acc, bar in zip(accuracys, bars):

    h = bar.get\_height() + 0.01

    x, y = bar.get\_x(), bar.get\_width()

    plt.text(x + y / 2, h, f"{acc:.3}", ha="center", va="bottom")

plt.xlabel("数字")

plt.ylabel("准确率")

plt.title("各个数字的准确率")

plt.xticks(tags)

# plt.grid(True)

plt.show()

### 在4.1.2的基础加上降维操作：mnist\_with\_pca.py

import numpy as np

import matplotlib.pyplot as plt

import utils

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score

from sklearn.decomposition import PCA

from matplotlib import rcParams

# 设置中文字体

rcParams["font.family"] = "SimHei"

# 加载数据

X, y, \_ = utils.load\_data("data/mnist.csv", 784, 784)

y = np.array(y, dtype=int)

# 划分数据集为训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, y, test\_size=0.3, random\_state=42

)

# 测试降维的维度(5, 15, 25...)

dimensions = range(5, 200, 10)

# 记录准确率

accuracys = []

# 记录最高的准确率以及使用的参数（准确率，目标维度）

best\_prama = (-1.0, -1)

for i in utils.tqdm(dimensions):

    # 指定降维的维度

    pca = PCA(n\_components=i)

    # 计算投影矩阵W

    pca.fit(X\_train)

    W = pca.components\_.T

    # 通过投影矩阵将当前训练集降维

    X\_train\_pca = X\_train @ W

    # 初始化朴素贝叶斯分类器

    naive\_bayesian = GaussianNB()

    # 训练模型

    naive\_bayesian.fit(X\_train\_pca, y\_train)

    # 对测试数据集降维

    X\_test\_pca = X\_test @ W

    # 预测测试集

    y\_pred = naive\_bayesian.predict(X\_test\_pca)

    # 计算准确率

    accuracy = accuracy\_score(y\_test, y\_pred)

    if accuracy > best\_prama[0]:

        best\_prama = (accuracy, i)

    accuracys.append(accuracy)

    # print(f"降到 {i} 维时，准确率为 {accuracy:.4}")

print(f"最佳维度为{best\_prama[1]}维，准确率为{best\_prama[0]}")

plt.figure(figsize=(12, 6))

# 画出折线图

plt.plot(dimensions, accuracys, marker="o", linestyle="-")

# 在每个数据点上显示具体数字

for i, (x, y) in enumerate(zip(dimensions, accuracys)):

    plt.text(x + 0.0008, y + 0.0015, f"{y:.3}", fontsize=10, ha="right")

plt.title("维度和准确率的关系")

plt.xlabel("维度")

plt.ylabel("准确率")

plt.grid(True)

plt.show()

# 测试最佳的维度

pca = PCA(n\_components=best\_prama[1])

# 计算投影矩阵W

pca.fit(X\_train)

W = pca.components\_.T

# 通过投影矩阵将当前训练集降维

X\_train\_pca = X\_train @ W

# 初始化朴素贝叶斯分类器

naive\_bayesian = GaussianNB()

# 训练模型

naive\_bayesian.fit(X\_train\_pca, y\_train)

# 对测试数据集降维

X\_test\_pca = X\_test @ W

# 预测测试集

y\_pred = naive\_bayesian.predict(X\_test\_pca)

# 计算每个数字的准确率

tags = np.unique(y\_test)

accuracys = []

for digit in tags:

    idx = y\_test == digit

    accuracys.append(accuracy\_score(y\_test[idx], y\_pred[idx]))

tag\_num = len(tags)

# 多分类混淆矩阵，行为预测值，列为真实值

confusion\_matrix = np.zeros((tag\_num, tag\_num), dtype=int)

# 记录每个分类结果的真实值和预测值

for real, pred in zip(y\_test, y\_pred):

    confusion\_matrix[pred][real] += 1

# 查准率

precision = confusion\_matrix.diagonal() / np.sum(confusion\_matrix, axis=1)

# 查全率

recall = confusion\_matrix.diagonal() / np.sum(confusion\_matrix, axis=0)

# F1

f1 = 2 \* np.multiply(precision, recall) / (precision + recall)

# 展示结果（混淆矩阵，查准率，查全率和F1\_Score)

utils.display(confusion\_matrix, precision, recall, f1, tags)

# 画出准确率图表

plt.figure(figsize=(12, 6))

bars = plt.bar(tags, accuracys)

# 在每个柱子上方添加准确率

for acc, bar in zip(accuracys, bars):

    h = bar.get\_height() + 0.01

    x, y = bar.get\_x(), bar.get\_width()

    plt.text(x + y / 2, h, f"{acc:.3}", ha="center", va="bottom")

plt.xlabel("数字")

plt.ylabel("准确率")

plt.title("各个数字的准确率")

plt.xticks(tags)

plt.show()

### 在4.1.3的基础加上平滑操作：mnist\_with\_smoothing.py

import numpy as np

import matplotlib.pyplot as plt

import utils

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score

from sklearn.decomposition import PCA

from matplotlib import rcParams

# 设置中文字体

rcParams["font.family"] = "SimHei"

# 设置降维的维度，0为不降维

pca = 75

# 加载数据

X, y, \_ = utils.load\_data("data/mnist.csv", 784, 784)

y = np.array(y, dtype=int)

# 划分数据集为训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, y, test\_size=0.3, random\_state=42

)

# 如果降维则对数据集进行操作

if pca != 0:

    pca = PCA(n\_components=pca)

    # 得到投影矩阵

    pca.fit(X\_train)

    W = pca.components\_.T

    # 通过投影矩阵将当前训练集和测试集降维

    X\_train = X\_train @ W

    X\_test = X\_test @ W

smoothings = np.linspace(1e-7, 5e-3, 40)

accuracys = []

for i in utils.tqdm(smoothings):

    # 初始化朴素贝叶斯分类器

    naive\_bayesian = GaussianNB(var\_smoothing=i)

    # 训练模型

    naive\_bayesian.fit(X\_train, y\_train)

    # 预测测试集

    y\_pred = naive\_bayesian.predict(X\_test)

    # 计算测试集准确率

    accuracy = accuracy\_score(y\_test, y\_pred)

    accuracys.append(accuracy)

    # print(f"alpha为{i}的时候，测试集上的准确度{accuracy}")

idx = np.argmax(accuracys)

print(f"当aplha={smoothings[idx]:.3}时，最佳准确度为{accuracys[idx]:.3}")

plt.figure(figsize=(15, 6))

# 画出折线图

plt.plot(smoothings, accuracys, marker="o", linestyle="-")

# 在每个数据点上显示具体数字

for i, (x, y) in enumerate(zip(smoothings, accuracys)):

    plt.text(x, y, f"{y:.3}", fontsize=10, ha="right")

plt.title("平滑系数和准确率的关系")

plt.xlabel("alpha")

plt.ylabel("准确率")

plt.grid(True)

plt.show()

### 工具类：utils.py

import csv

import numpy as np

from tqdm import tqdm

from tabulate import tabulate

def load\_data(path, num\_attr=4, tag\_col=4):

    print("loading data...")

    x, y = [], []

    with open(path, "r") as file:

        reader = csv.reader(file)

        title = next(reader)

        for row in tqdm(reader):

            x.append(np.array(row[:num\_attr], dtype=float))

            y.append(row[tag\_col])

    x = np.array(x)

    y = np.array(y)

    return x, y, title[:num\_attr]

# 展示结果（混淆矩阵，查准率，查全率和F1\_Score)

def display(cm, pc, rc, f1, tags):

    str = "\t" \* (len(tags) // 2 - 1)

    print(f"{str}混淆矩阵")

    headers = ["预测\实际"]

    data = []

    for x in tags:

        headers.append(x)

    for idx, tag in enumerate(tags):

        row = [tag] + cm[idx].tolist()

        data.append(row)

    print(tabulate(data, headers=headers))

    print("\n\t\t性能数据")

    headers = ["类型", "查准率", "查全率", "F1\_Score"]

    data = []

    for idx, tag in enumerate(tags):

        row = [tag, pc[idx], rc[idx], f1[idx]]

        data.append(row)

    print(tabulate(data, headers=headers))