模式识别实验报告

专业：	人工智能
学号：	58119314
年级：	2019
姓名：	马添毅

签名：

时间：

实验一 KNN分类任务

1.问题描述

概述

利用KNN算法对输血服务中心数据集中的测试集进行分类

数据集描述

输血服务中心数据集是UCI上的公开数据集。数据集包含多名献血者的信息 如最近一次献血到现在的时间跨度，献血总次数，献血总量，以及首次献血 到现在的时间跨度。数据集的相关信息如表1所示：

表1 输血服务中心数据集信息

样例数量	特征维度	特征类型	类别数量
798	4	数值	2

数据集已被划分为训练集、验证集和测试集，分别存储于data文件夹中的train_data.csv, val_data.csv, test_data.csv。train_data.csv 和val_data.csv文件包含data, label字段，分别存储着特征 $\mathbf{X} \in \mathbb{R}^{N \times d}$和标记 $\mathbf{Y} \in \mathbb{R}^{N \times 1}$。其中，N是样例数量，d = 4 为特征维度，每个样例的标记y ∈ { 0 , 1 }。test_data.csv 文件仅包含data字段。

任务说明

任务一

利用 **欧式距离、切比雪夫距离、曼哈顿距离** 作为KNN算法的度量函数 **测试集** 进行分类。实验报告中，要求分析三种距离度量在该数据集上的优劣。同时，要求在验证集上分析近邻数k对KNN算法分类精度的影响。

任务二

利用 **马氏距离** 作为KNN算法的度量函数，对 **测试集** 进行分类。

2.实现步骤及流程

读取数据

将训练集、验证集和测试集中csv格式的数据分别读取为dataframe，并对数据和标签进行分割和存储。

欧氏距离、切比雪夫距离、曼哈顿距离

1. 初始化分类器，根据不同距离度量的选择初始化为不同的分类器模型。（这三种距离度量都没有起到训练模型的作用）
2. 遍历验证集做标签预测：对于每个数据，选取其周围最近的（即特定距离度量下值最小的）训练集中的k个数据的标签，通过majority vote的方式决定验证集中数据的标签。

3. 通过与验证集实际标签的比对得到准确率。
4. 对不同的k值（奇数值）进行遍历，即可得到KNN模型在三种距离度量下、不同k值下各自的准确率。并且通过对准确率的排序可是最优及次优的k值。
5. 选取最优（实际选择了正确率次优值，原因为会结论部分详细说明）的k值对测试集中的数据进行标签预测。

马氏距离

马氏距离是一种可学习的度量函数，定义如下：

$$d_M\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right)=\sqrt{\left(\boldsymbol{x}_i-\boldsymbol{x}_j\right)^T \boldsymbol{M}\left(\boldsymbol{x}_i-\boldsymbol{x}_j\right)}$$

其中，$\mathbf{M} \in \mathbb{R}^{d \times d}$是一个半正定矩阵，是可以学习的参数。由于$\mathbf{M}$的半正定性质，可将上述定义表述为：

$$d_M\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right)=\sqrt{\left(\boldsymbol{x}_i-\boldsymbol{x}_j\right)^T \boldsymbol{A}^T \boldsymbol{A}\left(\boldsymbol{x}_i-\boldsymbol{x}_j\right)}=\left\|\boldsymbol{A} \boldsymbol{x}_i-\boldsymbol{A} \boldsymbol{x}_j\right\|_2$$

其中，矩阵 $\mathbf{A} \in \mathbb{R}^{e \times d}$。故**马氏距离**可以理解为对原始特征进行线性映射，然后计算**欧式距离**。

给定以下目标函数，在训练集上利用**梯度下降法**马氏距离进行学习：

$$\max_{\boldsymbol{A}} f(\boldsymbol{A})=\max_{\boldsymbol{A}} \sum_{i=1}^N \sum_{j \in C_i} p_{ij}$$

其中，C_i表示与样例x_i同类的样例集合，p_{ij}定义为：

$$p_{ij}=\begin{cases}\frac{\exp\left(-d_M\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right)^2\right)}{\sum_{k \neq i} \exp\left(-d_M\left(\boldsymbol{x}_i, \boldsymbol{x}_k\right)^2\right)} & j \neq i \\ 0 & j=i\end{cases}$$

实验中，矩阵$\mathbf{A}$的维度e设置为任一合适值。这里e=2。

经计算得到目标函数的梯度如下:

$$\begin{aligned}\frac{\partial f(A)}{\partial A} & =\sum_{i=1}^N \sum_{j \in C_i}\left(\frac{e^{-d_M\left(x_i, x_j\right)^2} \frac{\partial d_M\left(x_i, x_j\right)^2}{\partial A}}{\sum_{k \neq i} e^{-d_M\left(x_i, x_k\right)^2}}-\frac{e^{-d_M\left(x_i, x_j\right)^2} \sum_{k \neq i} e^{-d_M\left(x_i, x_k\right)^2} \frac{\partial-d_M\left(x_i, x_k\right)^2}{\partial A}}{\left(\sum_{k \neq i} e^{-d_M\left(x_i, x_k\right)^2}\right)^2}\right) \\ & =\sum_{i=1}^N \sum_{j \in C_i} p_{ij}\left[-2\left(x_i-x_j\right)^{\top} A\left(x_i-x_j\right)\right]-p_{ij} \sum_{k \neq i}\left[-2\left(x_i-x_k\right)^{\top} A\left(x_i-x_k\right)\right] \\ & =-2 A \sum_{i=1}^N \sum_{j \in C_i} p_{ij}\left[\left(x_i-x_j\right)^{\top}\left(x_i-x_j\right)-\left(x_i-x_k\right)^{\top}\left(x_i-x_k\right)\right]\end{aligned}$$

数学分析到这里就告一段落了。由以上的内容，在马氏距离下的KNN方法思路就很明朗了。

1. 数据处理同以上三种距离度量。但是需要注意对数据进行额外的归一化。
2. 随机初始化A矩阵。定义马氏距离，定义概率p矩阵。
3. 训练马氏距离的模型：计算梯度，根据梯度下降的方法更新A矩阵，直到loss function收敛（实际操作中还设置了最高的epoch数）。
4. 通过训练集和验证集选取最优的k值以及在测试集上做预测的工作和以上三种距离度量类似，便不再赘述。

3.实验结果与分析

k值对准确率的影响

需要说明的是，实验的数据有特殊性，因此分析也是围绕该实验下所得的结果展开的。比如k=1时的正确率远优于其他情况，猜测应该是模型产生了overfitting，因此在选择最优k值测试时，不选择正确率最好的k=1，而是在bias-variance trade-off只有，正确率的次优值下k的取值。当然，这个取值的合理性也未进行严谨地证明。

从四种距离下正确率随k变化的折线图中不难发现：当k值增加之后，总体上来说四种距离度量的模型准确率都是呈下降趋势的。

欧氏距离

最优k值为1，准确率高达93.2%；次优k值为3，准确率为81.1%。k增大后准确率收敛到68.9%。

切比雪夫距离

最优k值为1，准确率为87.8%；次优k值为5，准确率为79.7%。k增大后准确率收敛到68.9%。切比雪夫距离比较特别，在k=3、k=7、k=19的时候都有突然下降。分析原因可能是在这三个参数下计算出的距离不同，与代比较数据最近的数据在其他距离度量下不同。

曼哈顿距离

最优k值为1，准确率高达94.6%；次优k值为3，准确率为81.1%。k增大后准确率收敛到68.9%。

马氏距离

设置学习率为0.3，判断收敛的阈值为0.03。在6个epoch之后，loss function收敛到316.78。训练结束

```
objective: 316.0701283389266
Epoch 1/20       Gradient = [[-0.10878692  0.07262347  0.07262347  0.65330773]
 [-0.04285819  0.04903509  0.04903509  0.4404927 ]]
Epoch 1/20       objective: 316.2690457354537
Epoch 2/20       Gradient = [[-0.11759561  0.06010202  0.06010202  0.67225213]
 [-0.04863973  0.04322708  0.04322708  0.45438069]]
Epoch 2/20       objective: 316.4717260894882
Epoch 3/20       Gradient = [[-0.12679171  0.09383961  0.09383961  0.6361297 ]
 [-0.05438922  0.06735665  0.06735665  0.42860924]]
Epoch 3/20       objective: 316.6470985655438
Epoch 4/20       Gradient = [[-0.12967502  0.11105382  0.11105382  0.48795755]
 [-0.05466646  0.07857257  0.07857257  0.32783316]]
Epoch 4/20       objective: 316.74143981646444
Epoch 5/20       Gradient = [[-0.12286513  0.06998883  0.06998883  0.24624666]
 [-0.04759768  0.04928039  0.04928039  0.16575638]]
Epoch 5/20       objective: 316.76922371765437
Epoch 6/20       Gradient = [[-0.11717795  0.04458818  0.04458818  0.11136589]
 [-0.0423102   0.03127433  0.03127433  0.07506297]]
Epoch 6/20       objective: 316.7792897452006
```

最优k值为1，准确率高达91.9%；次优k值为3，准确率为81.1%。k增大后准确率收敛到68.9%。结果与以上三种距离度量相似。

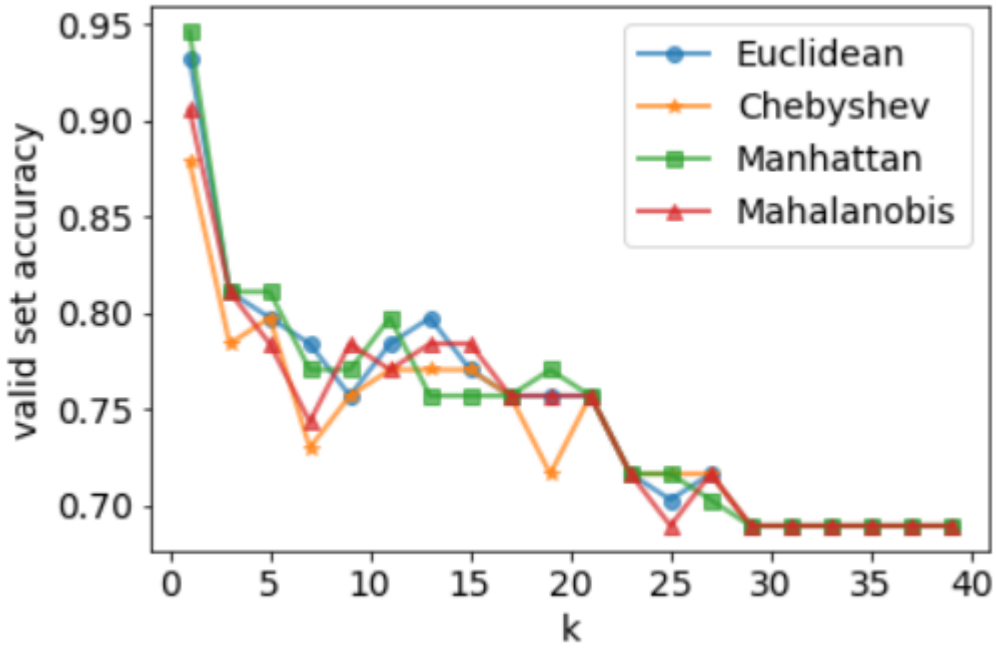


图1 四种距离下正确率随k变化的折线图

三种距离度量在该数据集上的优劣

欧氏距离在这个数据集维度不高的情况下，依然能有不错的表现，算作是其优点之一。计算复杂的不高是其优点之二。若是数据集维度变得更高的话，欧氏距离的效果可能就不尽如人意了。

切比雪夫距离在该数据集下表现并不稳定。原因可能在于本身这个距离度量就有适用的局限：更适合在物流等应用下，计算移动的距离。

曼哈顿距离看似是表现最好的，但它可能反映的不是数据之间的最短路径，可能给出比欧氏距离更大的距离值。因而在曼哈顿距离下的最近邻数据，未必是真正最接近待测数据的。

4.代码附录

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

#KNN分类器
class KNNClassifier():


    #初始化分类器
        #:param k: positive integer K最近邻法中K的取值
        #:param dist: "Euclidean" or "Mahalanobis"
    def __init__(self, k, dist="Euclidean", mat_dim=2):

        if isinstance(k, int) and k > 0:
            self.k = k
        else:
```

```
            raise ValueError("k必须是正整数，现在k是{}，类型为{}".format(k, type(k))) #对初始化K提出修改
要求

        if dist == "Euclidean" or dist == "Mahalanobis" or dist == "Chebyshev" or dist ==
"Manhattan" :
            self.dist = dist
        else:
            raise ValueError("距离度量不在本题的使用范围内") #对初始化distance提出修改要求

        if dist == "Mahalanobis":
            self.mat_dim = mat_dim


    # 出于数值精度考虑，需要先归一化，从数学上没有影响，但需要在所有位置都归一化
        #:param train_data: ndarray of n_train x d
        #:return normalized: max-min normalized data
    def data_normalize(self, train_data):

        normalized = np.empty_like(train_data)

        self.min_vals = train_data.min(axis=0)  # 保存为类变量，方便test time调用
        self.max_vals = train_data.max(axis=0)

        for j in range(train_data.shape[1]):
            span = self.max_vals[j] - self.min_vals[j]
            for i in range(train_data.shape[0]):
                normalized[i, j] = (train_data[i, j] - self.min_vals[j]) / span

        return normalized


    # 马氏距离
        #:param a: array
        #:param b: array of the same size
        #:return Mahalanobis distance between a and b
    def __dist_Mahalanobis(self, a, b):

        if len(a) != len(b):
            raise ValueError("两向量维度不一致")
#         if len(a) != self.A.shape[1]:
#             raise ValueError("马氏距离没有基于数据学习")

        n_features = len(a)
        a = a.reshape((n_features, 1))
        b = b.reshape((n_features, 1))

        dist_mah = np.sqrt((a - b).T @ self.A.T @ self.A @ (a - b))
        return dist_mah

    def __calculate_pro(self, train_data):
        n_data = train_data.shape[0]
        self.pro = np.zeros((n_data, n_data))  # 用一个矩阵存所有的p_ij
```

```
    for i in range(n_data):
        x_i = train_data[i]
        denominator = 0

        for j in range(n_data):
            if j != i:
                x_j = train_data[j]
                self.pro[i, j] = np.exp(-np.power(self.__dist_Mahalanobis(x_i, x_j),2))
                denominator += self.pro[i, j]

        self.pro[i, :] /= denominator

 # 马氏距离的学习的目标函数
    #:param train_data: ndarray of n_train x d
    #:param train_label: ndarray of n_train x 1
    #:return: output of the objective function
def __objective(self, train_data, train_label):

    total = 0
    for i in range(train_data.shape[0]):
        label = train_label[i]  # 得到一行数据对应的label
        index_of_this_class = np.argwhere(train_label == label)[:, 0]  # 得到同类的所有样本的索引

        for j in index_of_this_class:
            if j != i:
                total += self.pro[i, j] # f(A)= \sum \sum p_ij

    return total


# 学习马氏距离度量
    #:param train_data: ndarray of n_train x d
    #:param train_label: ndarray of n_train x 1
    #:param lr: learning rate, float
    #:param n_iter: number of iterations to conduct, int
    #:param early_stop: stop according to criterion, bool
    #:return loss_list
def fit(self, train_data, train_label, lr, n_iter, early_stop=True):

    if self.dist == "euclidean" or self.dist == "Chebyshev" or self.dist == "Manhattan":
        print("欧式距离、切比雪夫距离、曼哈顿距离不能被学习")
        return

    train_data = self.data_normalize(train_data)  # 归一化

    n_features = train_data.shape[1] #shape[1]为数据的列数
    self.A = np.random.rand(self.mat_dim, n_features)  # 随机初始化A矩阵
    self.__calculate_pro(train_data)  # 计算P矩阵
    loss_list = [self.__objective(train_data, train_label)]

    print("开始训练……")
```

```
        print("objective:", loss_list[0])

        # 计算梯度
        for current_iter in range(1, n_iter + 1):
            # print(f"Epoch {current_iter} starts")

            temp = np.zeros((n_features, n_features))
            for i in range(train_data.shape[0]):
                x_i = train_data[i]
                label = train_label[i]  # 得到一行数据对应的label
                index_of_this_class = np.argwhere(train_label == label)[:, 0]  # 得到同类的所有样本的索引

                for j in index_of_this_class:
                    if j != i:
                        subtemp = np.zeros((n_features, n_features))
                        for k in range(train_data.shape[0]):
                            if k != i:
                                x_k = train_data[k]
                                x_ik = (x_i - x_k).reshape((n_features, 1))
                                subtemp += self.pro[i, k] * (x_ik @ x_ik.T) # \sum p_ij@(x_i-x_k)^T@(x_i-x_k)

                        x_j = train_data[j]
                        x_ij = (x_i - x_j).reshape((n_features, 1))
                        temp += self.pro[i, j] * ((x_ij @ x_ij.T) - subtemp)

            grad = - 2 * self.A @ temp
#             print("Epoch {}/{} \t Gradient = {}".format(current_iter, n_iter, grad))
            self.A += lr * grad  # 梯度上升
            self.__calculate_pro(train_data)  # 更新P矩阵
            loss_list.append(self.__objective(train_data, train_label))
            print(f"Epoch {current_iter}/{n_iter}\t objective: {loss_list[-1]}")

            # 提前中止
            if early_stop and current_iter > 5:
                if abs(loss_list[-1] - loss_list[-2]) < 0.03: #threshold设置
                    break

        return loss_list
    #To do: How to 更高效地计算p_ij，每次更新A后都必须重新计算!


    # 预测单个样本
        #:param train_data: nd-array of n_train x d
        #:param train_label: nd-array of n_train x 1
        #:param test_ins: 矩阵维度 d-dimensional array
        #:return test_label: integer
    def predict(self, train_data, train_label, test_ins):

#         train_data = self.data_normalize(train_data) #训练样本归一化
#         test_ins = (test_ins - self.min_vals) / (self.max_vals - self.min_vals)  # 测试样本初始化
```

```
        if self.dist == "Euclidean":
            distances = np.linalg.norm(train_data - test_ins, axis=1)  #欧氏距离赋值，ord=None 默认为L2-norm；axis=1表示按行向量处理，求多个行向量的范数；

        if self.dist == "Chebyshev":
            distances = np.linalg.norm(train_data - test_ins, ord = np.inf, axis=1)

        if self.dist == "Manhattan":
            distances = np.linalg.norm(train_data - test_ins, ord = 1, axis=1)

        if self.dist == "Mahalanobis":
            distances = np.empty(train_data.shape[0])  #马氏距离初始化
            for i in range(len(distances)):
                distances[i] = self.__dist_Mahalanobis(test_ins, train_data[i]) #马氏距离赋值

        train_label = train_label.flatten()
        k_neighbor_index = np.argsort(distances)[:self.k]
        neighbor_labels = train_label[k_neighbor_index]
        most_voted = np.argmax(np.bincount(neighbor_labels))
#         print("在k={}下，邻居点的标签为{}，最后判定的结果应当为{}".format(self.k, neighbor_labels, most_voted))

        return most_voted

k_values = range(1,41,2)
#测试欧式距离 trian vs valid
def test_Euclidean(train, valid):

    train_frames = [train['R'],train['F'],train['M'],train['T']]
    X_train = np.array(pd.concat(train_frames, axis = 1))
    Y_train = np.array(train['label'])

    valid_frames = [valid['R'],valid['F'],valid['M'],valid['T']]
    X_valid =  np.array(pd.concat(valid_frames, axis = 1))
    Y_valid =  np.array(valid['label'])

    cls = KNNClassifier(1)
    acc_list = []
    for k in k_values:
        cls.k = k
        true_cnt, total_cnt = 0, 0
        for i in range(X_valid.shape[0]):
            sample = X_valid[i]
            pred = cls.predict(X_train, Y_train, sample)#对X_valid进行预测
            truth = Y_valid[i]
            if pred == int(truth):
                true_cnt += 1
            total_cnt += 1

        acc_list.append(true_cnt / total_cnt)
```

```
    print("Euclidean distance:", acc_list)

    return cls, acc_list

#测试切比雪夫距离 trian vs valid
def test_Chebyshev(train, valid):

    train_frames = [train['R'],train['F'],train['M'],train['T']]
    X_train = np.array(pd.concat(train_frames, axis = 1))
    Y_train = np.array(train['label'])

    valid_frames = [valid['R'],valid['F'],valid['M'],valid['T']]
    X_valid =  np.array(pd.concat(valid_frames, axis = 1))
    Y_valid =  np.array(valid['label'])

    cls = KNNClassifier(1,dist="Chebyshev")
    acc_list = []
    for k in  k_values:
        cls.k = k
        true_cnt, total_cnt = 0, 0
        for i in range(X_valid.shape[0]):
            sample = X_valid[i]
            pred = cls.predict(X_train, Y_train, sample)
            truth = Y_valid[i]
            if pred == int(truth):
                true_cnt += 1
            total_cnt += 1

        acc_list.append(true_cnt / total_cnt)

    print("Chebyshev distance:", acc_list)

    return cls, acc_list

#测试曼哈顿距离 trian vs valid
def test_Manhattan(train, valid):

    train_frames = [train['R'],train['F'],train['M'],train['T']]
    X_train = np.array(pd.concat(train_frames, axis = 1))
    Y_train = np.array(train['label'])

    valid_frames = [valid['R'],valid['F'],valid['M'],valid['T']]
    X_valid =  np.array(pd.concat(valid_frames, axis = 1))
    Y_valid =  np.array(valid['label'])

    cls = KNNClassifier(1,dist="Manhattan")
    acc_list = []
    for k in  k_values:
        cls.k = k
        true_cnt, total_cnt = 0, 0
        for i in range(X_valid.shape[0]):
            sample = X_valid[i]
```

```
            pred = cls.predict(X_train, Y_train, sample)
            truth = Y_valid[i]
            if pred == int(truth):
                true_cnt += 1
            total_cnt += 1

        acc_list.append(true_cnt / total_cnt)

    print("Manhattan distance:", acc_list)

    return cls, acc_list

#测试马氏距离 trian vs valid
def test_Mahalanobis(train, valid):

    train_frames = [train['R'],train['F'],train['M'],train['T']]
    X_train = np.array(pd.concat(train_frames, axis = 1))
    Y_train = np.array(train['label'])

    valid_frames = [valid['R'],valid['F'],valid['M'],valid['T']]
    X_valid =  np.array(pd.concat(valid_frames, axis = 1))
    Y_valid =  np.array(valid['label'])

    acc_list = []
    cls = KNNClassifier(1, dist="Mahalanobis", mat_dim=2)
    loss = cls.fit(X_train, Y_train, lr=0.3, n_iter=20) #学习马氏距离，参数还有待调整

    for k in k_values:
        cls.k = k
        true_cnt, total_cnt = 0, 0
        for i in range(X_valid.shape[0]):
            sample = X_valid[i]
            pred = cls.predict(X_train, Y_train, sample)
            truth = Y_valid[i]
            if pred == int(truth):
                true_cnt += 1
            total_cnt += 1

        acc_list.append(true_cnt / total_cnt)

    print("Mahalanobis distance:", acc_list)

    return cls, acc_list


# k_values = range(1,21)
import pickle #仅用于保存模型，对题目没有帮助

# k_values = range(1,21)
import pickle #仅用于保存模型，对题目没有帮助

def main():
```

```
    # 读取数据
    train = pd.read_csv("./train_data.csv", header = 0, names = ['R','F','M','T','label'])
    valid = pd.read_csv("./val_data.csv", header = 0, names = ['R','F','M','T','label'])
    test = pd.read_csv("./test_data.csv", header = 0, names = ['R','F','M','T'])

    #转变数据类型，提升速度
    train_frames = [train['R'],train['F'],train['M'],train['T']]
    X_train = np.array(pd.concat(train_frames, axis = 1))
    Y_train = np.array(train['label'])

    valid_frames = [valid['R'],valid['F'],valid['M'],valid['T']]
    X_valid =  np.array(pd.concat(valid_frames, axis = 1))
    Y_valid =  np.array(valid['label'])

    test_frames = [test['R'],test['F'],test['M'],test['T']]
    X_test =  np.array(pd.concat(test_frames, axis = 1))


    k_value_array = np.array(k_values)
    k_value_list = k_value_array.tolist()

    #以欧氏距离为度量的模型
    cls_euc, euc_acc_list = test_Euclidean(train, valid)
    k_value_pair_euc = list(zip(k_value_list,euc_acc_list))
    cls_euc.k = sorted(k_value_pair_euc, key = lambda x:x[1], reverse = True)[1][0]
#     print(k_value_pair_euc)
    print("Euclidean下 次优的k值：{}".format(cls_euc.k))
    pickle.dump(cls_euc,open("KNNClassifier_Euclidean.dat","wb")) #保存模型
#     cls_euc_1 = pickle.load("KNNClassifier_Euclidean.dat")

    #以切比雪夫距离为度量的模型
    cls_che, che_acc_list = test_Chebyshev(train, valid)
    k_value_pair_che = list(zip(k_value_list,che_acc_list))
    cls_che.k = sorted(k_value_pair_che, key = lambda x:x[1], reverse = True)[1][0]
#     print(k_value_pair_che)
    print("Chebyshev下 次优的k值：{}".format(cls_che.k))
    pickle.dump(cls_che,open("KNNClassifier_Chebyshev.dat","wb")) #保存模型
#     print(sorted(k_value_pair_che, key = lambda x:x[1], reverse = True))

    #以曼哈顿距离为度量的模型
    cls_man, man_acc_list = test_Manhattan(train, valid)
    k_value_pair_man = list(zip(k_value_list,man_acc_list))
    cls_man.k = sorted(k_value_pair_man, key = lambda x:x[1], reverse = True)[1][0]
#     print(k_value_pair_man)
    print("Manhattan下 次优的k值：{}".format(cls_man.k))
    pickle.dump(cls_man,open("KNNClassifier_Manhattan.dat","wb")) #保存模型


#     使用Chebyshev距离预测test数据
    with open('temp1.csv','w',encoding = 'utf-8') as f:
        f.write("My prediction\n")
        for i in range(X_test.shape[0]):
```

```
            test_ins = X_test[i]
            pred = cls_che_1.predict(X_train, Y_train, test_ins)
            f.write("{:d}\n".format(pred))

    predict_data_1 = pd.read_csv('temp1.csv')
    test_data = pd.read_csv('test_data.csv')
    test_data['My prediction'] = predict_data_1
    test_data.to_csv('task1_test_prediction.csv', mode = 'a', index =False)

    #以马氏距离为度量的模型
    cls_mah, mah_acc_list = test_Mahalanobis(train, valid)
    k_value_pair_mah = list(zip(k_value_list,mah_acc_list))
    cls_mah.k = sorted(k_value_pair_mah, key = lambda x:x[1], reverse = True)[1][0]
    print("Mahalanobis下 次优的k值：{}".format(cls_mah.k))

    #保存模型
    with open('./KNNClassifier_Mahalanobis.dat','wb') as f:
        pickle.dump(cls_mah, f)
#     #调取模型
#     with open('./KNNClassifier_Mahalanobis.dat','rb') as f:
#         cls_mah_1 = pickle.load(f)

    #存储数据
     with open('temp2.csv', 'w', encoding='utf-8') as f:
        f.write("My prediction\n")
        for i in range(X_test.shape[0]):
            test_ins = X_test[i]
            pred = cls_mah_1.predict(X_train, Y_train, test_ins)
            f.write("{:d}\n".format(pred))

    predict_data_2 = pd.read_csv('temp2.csv')
    test_data_2 = pd.read_csv('test_data.csv')
    test_data_2['My prediction'] = predict_data_1
    test_data_2.to_csv('task2_test_prediction.csv', mode = 'a', index =False)


    #绘制图像
    plt.figure()
    plt.plot(k_values, euc_acc_list, linewidth='2', marker='o',alpha = 0.7, label='Euclidean')
    plt.plot(k_values, che_acc_list, linewidth='2', marker='*',alpha = 0.7, label='Chebyshev')
    plt.plot(k_values, man_acc_list, linewidth='2', marker='s',alpha = 0.7, label='Manhattan')
    plt.plot(k_values, mah_acc_list, linewidth='2', marker='^',alpha = 0.7, label='Mahalanobis')
    plt.xlabel("k", fontsize=14)
    plt.ylabel("valid set accuracy", fontsize=14)
    plt.xticks([0, 5, 10, 15, 20, 25, 30, 35, 40], fontsize=14)
    plt.yticks(fontsize=14)
    plt.legend(fontsize=14)
    plt.show()


if __name__ == "__main__":
    main()
```

```
<div STYLE="page-break-after: always;"></div>
```

实验二 KNN分类任务

1.问题描述

概述

利用线性分类器对Kuzushiji-MNIST数据集中的测试集进行分类。

实验平台及数据说明

Kuzushiji-MNIST是古日文的手写体识别数据集。该数据集由训练数据集和测试数据集两部分组成，其中训练数据集包含了60,000张样本图片及其对应标签，每张图片由28×28的像素点构成；训练数据集包含了10,000张样本图片及其对应标签，每张图片由28×28的像素点构成。

任务说明

任务一

Kuzushiji-MNIST数据集进行预处理，然后在处理后的训练集上学习一个多类线性分类器，并对处理后的测试集进行分类。

任务二

利用PCA降维方法对Kuzushiji-MNIST数据集进行降维，然后在降维后的数据上完成多类线性分类器的训练和测试。要求比较应用PCA降维技术前后，分类器准确率的变化。（对于降维后的数据，可以尝试利用可视化方法展示结果。）

2.实现步骤及流程

设计分类器

本实验的核心问题是要实现一个多类线性分类器。大部分线性分类器原生都是针对二分类问题的，例如逻辑回归（Logistic Regression）、支持向量机（Support Vector Machine）、感知机（Perceptron）等。如要将这些模型应用于多分类任务，
则需要将多分类任务拆解为若干个二分类任务，常见策略有一对一（One vs. One）、一对多（One vs. Rest）等。然而，这些策略往往会在特征空间中留下一些不确定的区域。因此，我考虑能够原生适应多分类问题的分类器。

在本次实验中，采用SoftMax 分类器。输入任意一个样本$\mathbf{x} \in \mathbb{R}^d$，SoftMax 分类器会输出一个概率向量，其中的每个分量是该样本属于每一个类$w_i, i = 1, \ldots, c$的概率的预测值$p(y = w_i|\mathbf{x})$，选取预测概率最大的类别作为分类结果。

SoftMax分类器的前向预测过程可以表示成这样的判别函数：

$$
\begin{aligned}
h_\Theta(\boldsymbol{x}) = \boldsymbol{z} &= \begin{pmatrix} \Pr(y = w_1 \mid \boldsymbol{x}) \\ \vdots \\ \Pr(y = w_c \mid \boldsymbol{x}) \end{pmatrix} = \frac{1}{\sum_{j=1}^{c} e^{\theta_j^T \boldsymbol{x}}} \begin{pmatrix} e^{\theta_1^T \boldsymbol{x}} \\ \vdots \\ e^{\theta_c^T \boldsymbol{x}} \end{pmatrix} \\
g_i(\boldsymbol{x}) &= z_i, \quad i = 1, \cdots, c
\end{aligned}
$$

其中，需要学习的参数是矩阵$\Theta=\left(\theta_1^T,\ldots,\theta_c^T\right)^T\in\mathbb{R}^{c\times(d+1)}$，也包括偏置参数。采用的loss function是交叉熵（Cross Entropy）损失，其目的是为了让预测结果的概率分布拟合真实的概率分布$\mathbf{y}$，也即真实类的概率为1，其余类的概率为0.

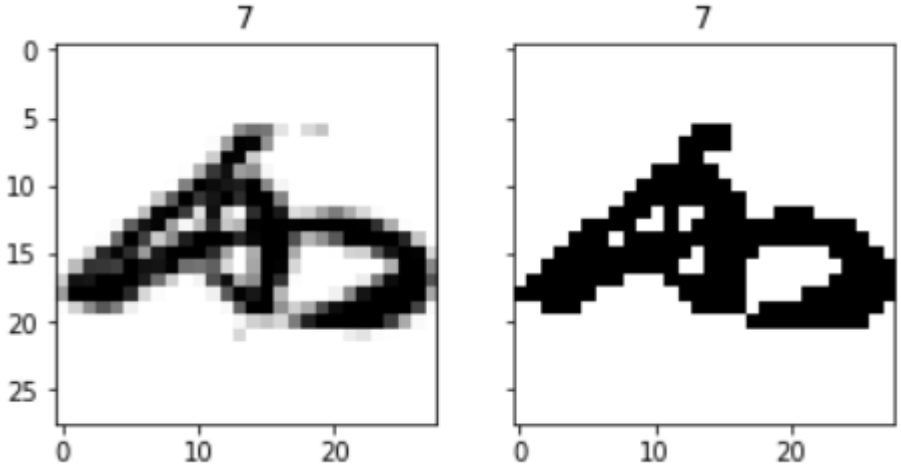
$$\begin{aligned}\widehat{\Theta}&=\underset{\Theta}{\operatorname{argmin}}\,L(\Theta)\\&=\underset{\Theta}{\operatorname{argmin}}\,\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{c}-y_j\log\Pr(y_i=w_j\mid\boldsymbol{x}_i;\Theta)\\&=\underset{\Theta}{\operatorname{argmin}}-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{c}1\left\{y_i=w_j\right\}\log h_\Theta(\boldsymbol{x})\end{aligned}$$

这个目标函数可以通过梯度下降的方式来最小化。计算交叉熵的梯度后，通过公式$\Theta\leftarrow\Theta-\mu\nabla_\Theta L(\Theta)$来更新参数，直至收敛：

$$\nabla_\ominus L(\Theta)=-\frac{1}{N}\sum_{i=1}^{N}\boldsymbol{x}_i(h_\ominus(\boldsymbol{x})-\boldsymbol{y}_i)^T$$

读取并处理数据

数据包括：训练集图像、训练集标签、测试集图像，对所有的图像进行二值化处理，将原图片的像素值映射为0和1，使得图片更加清晰。



对测试集进行分类

由于θ的维度是$c\times d+1$，因此需要对保存的二值化图像数组进行列方向上的增广（令$\theta_0=1$）。

PCA降维

由下面的公式，计算数据集的均值$\mathbf{m}$，得到$\mathbf{S}$矩阵。计算$\mathbf{S}$的特征值，当投影维度为10时，选取特征值最大的10个特征向量组成线性投影矩阵。

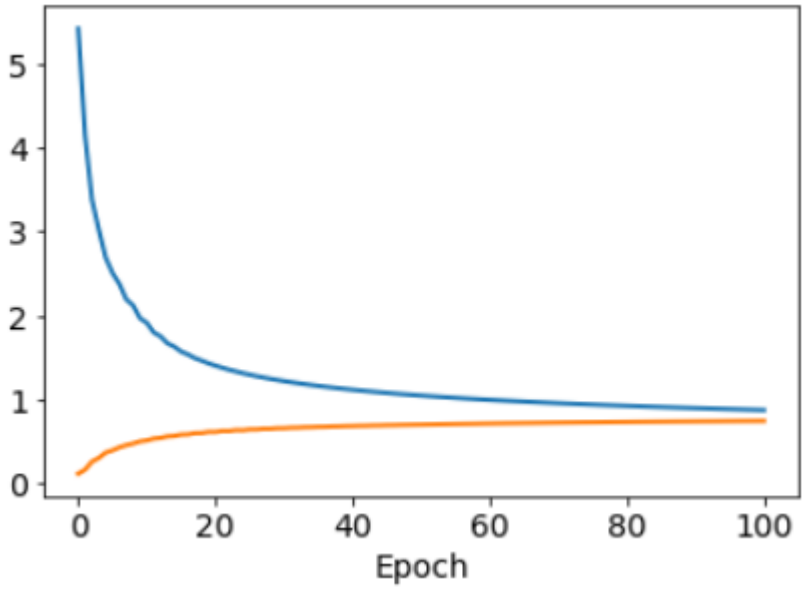
$$\begin{aligned}\mathbf{m}&=\frac{1}{n}\sum_{k=1}^{n}\mathbf{x}_k\\\mathbf{S}&=\sum_{k=1}^{n}(\mathbf{x}_k-\mathbf{m})(\mathbf{x}_k-\mathbf{m})^\top\end{aligned}$$

3.实验结果与分析

任务一

学习率为0.8，进行了100个epoch之后，loss收敛到了0.88，正确率也收敛到了74.7%。

当学习率设置地更低时，如0.3、0.5，那么模型的效果提升不明显；而当学习率设置地较大时，如3、5，则loss function在一开始会有明显的抖动。



预测结果详见表格 **task1_test_prediction.csv**

任务二

将数据矩阵用PCA投射到2维空间中，投射后的数据图如下所示。发现降维在边缘效果较好，在中心效果一般。

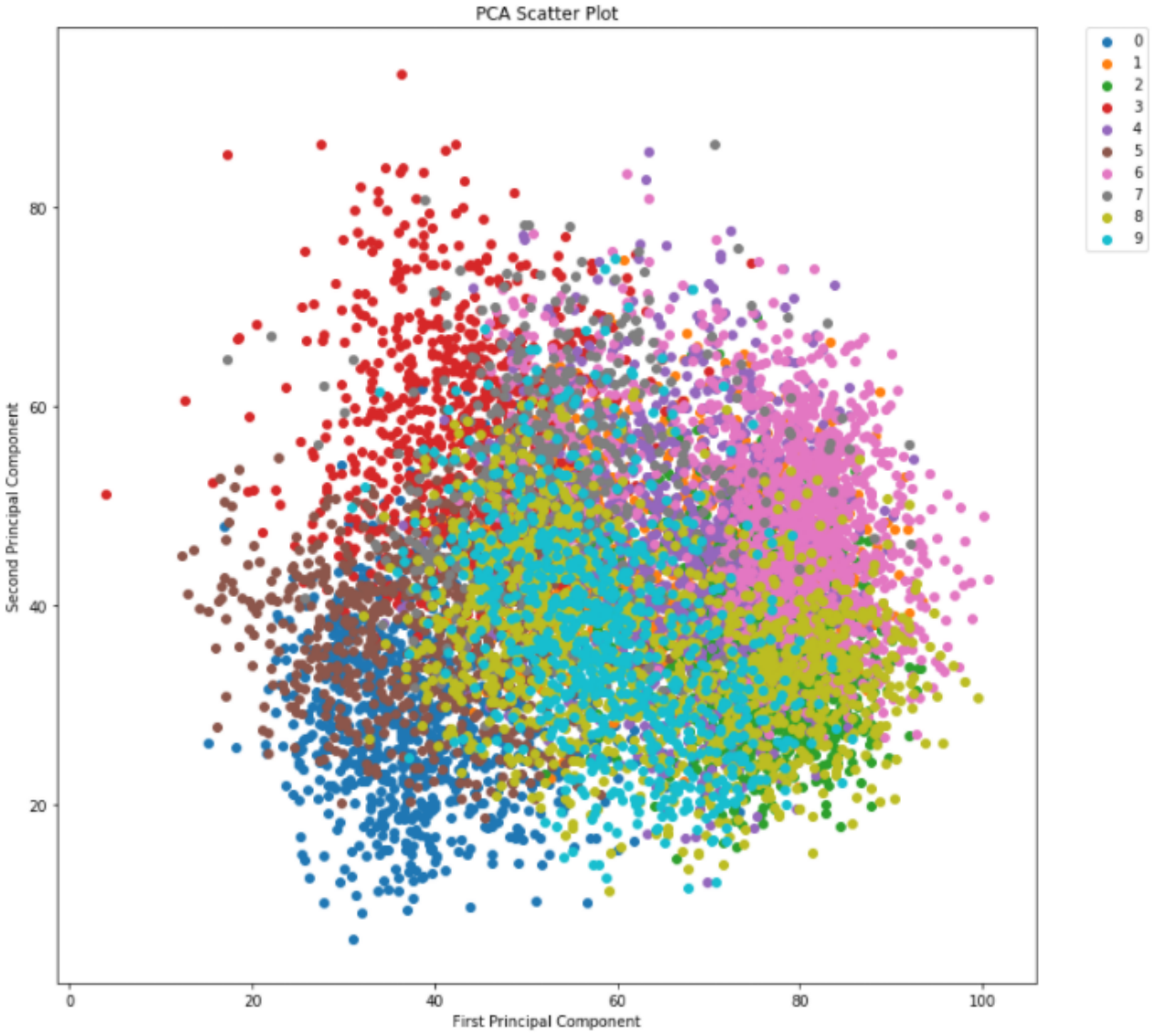
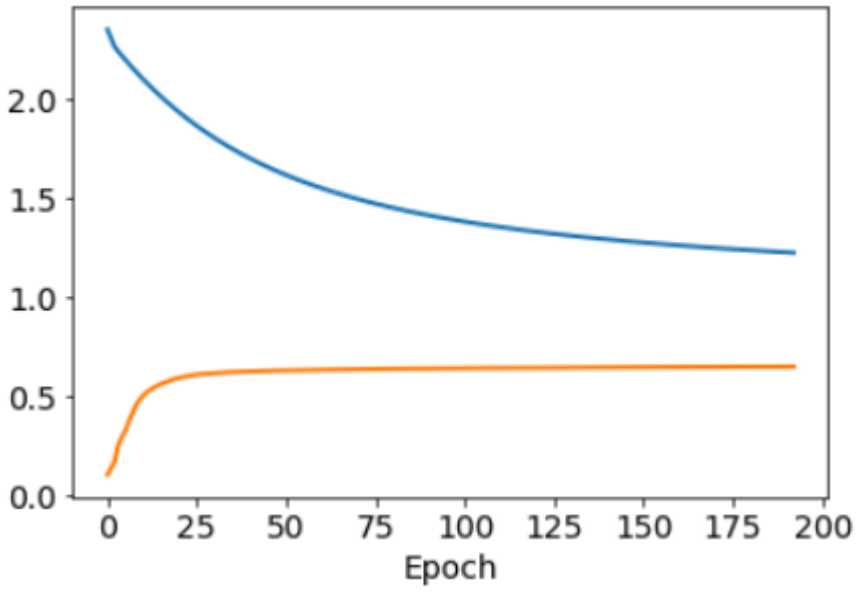


图2 PCA降维后的数据散点分布

学习率为5，进行了192个epoch之后（threshold=1e-3），loss收敛到了1.22，正确率也收敛到了64.56%。学习率为0.8，进行了353个epoch之后（threshold=1e-3），loss收敛到了1.56，正确率则收敛到了63.05%。

相比于未做降维之前的结果，略有下降。原因可能是降噪处理对于该数据集起到的效果不明显，数据集中的噪声对模型的预测还是产生了较大的干扰。



预测结果详见**task2_test_prediction.csv**

4.代码附录

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

#Softmax分类器
class SoftmaxClassifier:

    #初始化分类器
        #:param train_data: ndarray of n_data x n_feats
        #:param train_label: array of length n_data
        #:param n_classes: integer of n_classes
    def __init__(self, train_data, train_label, n_classes):

        if len(train_data.shape)>2:
            raise ValueError("需要先将训练数据变成二维矩阵")

        #加载数据集
        self.train_data = train_data

        #增广数据矩阵(多1列)
        n_data, n_features = train_data.shape
        self.train_data =np.ones((n_data, n_features+1), dtype = np.float)
        self.train_data[:,1:] = train_data #第一列是1，其他为原数据矩阵
        self.n_data, self.n_features = self.train_data.shape

        self.n_classes = n_classes

        #加载标签
        self.train_label = train_label

        #onehot编码
        self.one_hot = np.zeros((self.n_data, self.n_classes))
        for i in range(n_data):
            self.one_hot[i, train_label[i]] = 1

        #随机初始化W矩阵
        self.W = np.random.rand(self.n_classes, self.n_features)

        #打印检查
        print("n_classes:", self.n_classes)
        print("n_data:", self.n_data)
        print("n_feats", self.n_features)
        print("train_data.shape:", self.train_data.shape)
        print("train_label.shape:", self.train_label.shape)
        print("W.shape:", self.W.shape)


    #假设函数softmax
```

```
def softmax(self,x):

    #对单个样本向量
    if len(x.shape) == 1:
        numerator = np.exp(self.W @ x.reshape(-1, 1)).flatten()
        denominator = numerator.sum()
    #对样本矩阵
    else:
        numerator = np.exp(self.W @ x.T)
        denominator = numerator.sum(axis = 0)

    return numerator/denominator


#损失函数 cross entropy
def loss(self):

    softmax_matrix = self.softmax(self.train_data)
    total = 0
    for i in range(self.n_data):
        true_label = self.train_label[i]
        total += np.log(softmax_matrix[true_label, i])

    return -total/self.n_data


#对测试样本进行分类
def predict(self, test_data):

    probability = self.softmax(test_data)
    #对单个样本
    if len(probability.shape)<2:
        return np.argmax(probability)
    #对样本矩阵
    else:
        return np.argmax(probability, axis=0)


#训练正确率
def train_accuracy(self):
    predict_result = self.predict(self.train_data)
    n_true_predict = (predict_result == self.train_label).sum()
    return n_true_predict/self.n_data


#用SGD训练Softmax分类器
    #:param lr learning rate
def train(self, lr, n_epoch, early_stop=True):

    print("开始训练……")
    loss_list = [self.loss()]
    acc_list = [self.train_accuracy()]
```

```
        print(f"loss: {loss_list[-1]},train accuracy: {acc_list[-1]*100}%") #输出列表最后一个元素

        #使用梯度下降更新参数
        for current_epoch in range(1, n_epoch+1):
            print(f"Epoch {current_epoch} starts")

            #调整learning rate
            if current_epoch == 20:
                lr /= 2
            if current_epoch == 50:
                lr /= 2

            #计算梯度
            grad = 0
            for i in range(self.n_data):
                x_i = self.train_data[i]
                grad += np.outer(self.softmax(x_i)-self.one_hot[i],x_i)
            grad /= self.n_data
            #更新梯度
            self.W -= lr*grad

            loss_list.append(self.loss())
            acc_list.append(self.train_accuracy())
            print(f"loss: {loss_list[-1]},train accuracy: {acc_list[-1]*100}%") #输出列表最后一个元素

            if early_stop:
                if current_epoch > 0.2*n_epoch and abs(loss_list[-1]-loss_list[-2]) < 1e-3 \
                        and abs(loss_list[-2]-loss_list[-3]) <1e-3:
                    break

        return np.array(loss_list), np.array(acc_list)

#二值化
    #:param array: 数组
    #:param th: 阈值
    #:return: 二值化矩阵（0和1）
def binarize(array, threshold):
    return (array>threshold).astype(np.int8)

#针对二维矩阵进行归一化
def normalize(array):
    min_vals = array.min(axis = 0)
    max_vals = array.max(axis = 0)
    return (array - min_vals) / (max_vals - min_vals), min_vals, max_vals

#使用PCA分析降维数据
    #:param project_dim: the dimension of projected data
    #:return: the projection weight matrix
def PCA(train_data, train_label, project_dim = 2, n_classes =10):

    #检测数据维数是否正确
    if project_dim >= n_classes:
```

```
        raise ValueError("PCA的投影维数必须小于类数")

    n_data, n_features = train_data.shape

    m = np.average(train_data, axis = 0)

    S = np.zeros((n_features, n_features))

    for label in range(n_classes):
        index_of_class = np.argwhere(train_label == label) #得到所有同类样本的索引
        temp = np.zeros((n_features, n_features))
        for i in index_of_class:
            temp += np.outer(train_data[i]-m, train_data[i]-m)
#             print(np.outer(train_data[i]-m, train_data[i]-m).shape)
        S += temp


    eigenvalues, eigenvectors = np.linalg.eig(S) #获得S矩阵的特征值、特征向量

    eigen_index = np.argsort(eigenvalues)[::-1][:project_dim]

    W = eigenvectors[:, eigen_index].astype(float)

    return W

#读取训练集图像
with open("./train-images-idx3-ubyte", "rb") as f:
    f.read(4) #从文件当前位置起读取4个字节
    train_image_count = int.from_bytes(f.read(4),'big') #把bytes类型的变量转化为十进制整数，big为正常顺序
    row_count = int.from_bytes(f.read(4), 'big')
    column_count = int.from_bytes(f.read(4), 'big')
    train_image_data = f.read()  # 剩余的所有字节至文件结束
    train_images = np.frombuffer(train_image_data, dtype = np.uint8).reshape(
        (train_image_count, row_count * column_count))

#读取训练集标签
with open("./train-labels-idx1-ubyte", "rb") as f:
    f.read(8)
    label_data = f.read()
    train_labels = np.frombuffer(label_data, dtype = np.uint8)

#读取测试集图像
with open("./t10k-images-idx3-ubyte", "rb") as f:
    f.read(4) #从文件当前位置起读取4个字节
    test_image_count = int.from_bytes(f.read(4),'big') #把bytes类型的变量转化为十进制整数，big为正常顺序
    row_count = int.from_bytes(f.read(4), 'big')
    column_count = int.from_bytes(f.read(4), 'big')
    test_image_data = f.read()  # 剩余的所有字节至文件结束
    test_images = np.frombuffer(test_image_data, dtype = np.uint8).reshape(
        (test_image_count, row_count * column_count))
```

```
#二值化并保存
binary_train_images = binarize(train_images, 100) #threshold = 100
binary_test_images = binarize(test_images, 100)
np.save("binary-train-images.npy", binary_train_images)
np.save("binary-test-images.npy", binary_test_images)


#任务一_3.0
cls = SoftmaxClassifier(binary_train_images, train_labels, n_classes = 10)
loss_list, acc_list = cls.train(lr = 0.8, n_epoch = 100) # lr=0.5时提升效果不明显

plt.plot(loss_list, linewidth = '2', label = 'Loss')
plt.plot(acc_list, linewidth = '2', label = 'Accuracy')
plt.xticks(fontsize = 14)
plt.yticks(fontsize = 14)
plt.xlabel("Epoch", fontsize = 14)
# plt.ylabel("Loss", fontsize = 14)
plt.show()

augmented_binary_test_images = np.ones((test_image_count, row_count*column_count+1))
augmented_binary_test_images[:,1:] = binary_test_images
predict_result = cls.predict(augmented_binary_test_images)
print(predict_result)

with open('task1_test_prediction.csv', 'w', encoding = 'utf-8') as f:
    for i in range(test_image_count):
        f.write(f'./test{i}.jpg,{predict_result[i]}\n')


#任务二
PCA_projection_dim = 9

# #加载模型
# projection_matrix = np.load(f'{PCA_projection_dim}d_projection_matrix.npy')

projection_matrix = PCA(binary_train_images, train_labels, project_dim = PCA_projection_dim,
n_classes = 10)
print(projection_matrix)
#储存模型
np.save(f'{PCA_projection_dim}d_projection_matrix.npy',projection_matrix)


#对数据进行降维处理
pca_train_images = binary_train_images @ projection_matrix
normalized_pca_train_images, pca_min_vals, pca_max_vals = normalize(pca_train_images)

pca_test_images = binary_test_images @ projection_matrix
normalized_pca_test_images = (pca_test_images-pca_min_vals)/(pca_max_vals-pca_min_vals)
augmented_pca_test_images = np.ones((test_image_count, PCA_projection_dim +1))
augmented_pca_test_images[:, 1:] = normalized_pca_test_images

#加载模型
```

```python
cls = SoftmaxClassifier(normalized_pca_train_images, train_labels, n_classes = 10)
loss_list, acc_list = cls.train(lr = 5, n_epoch = 500) # lr=0.5时提升效果不明显

plt.plot(loss_list, linewidth = '2', label = 'Loss')
plt.plot(acc_list, linewidth = '2', label = 'Accuracy')
plt.xticks(fontsize = 14)
plt.yticks(fontsize = 14)
plt.xlabel("Epoch", fontsize = 14)
# plt.ylabel("Loss", fontsize = 14)
plt.show()

predict_result = cls.predict(augmented_pca_test_images)
print(predict_result)

with open('task2_test_prediction.csv', 'w', encoding = 'utf-8') as f:
    for i in range(test_image_count):
        f.write(f'./test{i}.jpg,{predict_result[i]}\n')
```

实验三 隐马尔科夫模型分词任务

1.问题描述

概述

利用隐马尔科夫模型进行中文语句的分词。

数据说明

数据集是人民日报1998年1月份的语料库，对600多万字节的中文文章加入了词性标注以及分词处理，由北京大学开发，是中文分词统计的常用资料。可以在语料库基础上构建词典、进行统计、机器学习等。

任务说明

中文信息处理是自然语言今天处理的分支，是指用计算机对中文进行处理。和大部分西方语言不同，书面汉语的词语之间没有明显的空格标记，句子是以字串的形式出现。因此对中文进行处理的第一步就是进行分词，即将字串转变成词串通过确立状态集合(B, M, E, S)，四个字母分别代表一个字在词语中的开始/中间/结尾/或者单字成词，这样可以将输入的中文句子编为一段状态序列，然后计算初始状态概率、转移概率及发射概率实现整个算法过程。

在人民日报分词语料库上统计语料信息，对隐马尔科夫模型进行训练。利用训练好的模型，对以下语句进行分词测试：

> 1) 今天我来到了东南大学。
> 2) 模式识别课程是一门有趣的课程。
> 3) 我认为完成本次实验是一个挑战。

2.实现步骤及流程

实验思路

观测值即是一句话中的每一个字。如果得到了一个句子（观测序列）对应的状态序列，即可利用状态进行分词。这样，分词问题就转化为了隐马尔科夫模型的解码问题，可以用Viterbi 算法求解。隐马尔科夫模型的学习即是从训练集中学习状态转移矩阵、观测。如果数据只有观测序列，那么需要利用Baum-Welch 算法进行学习。而本实验采用的人民日报数据集是已经分词完毕的，即既有观测序列也有状态序列。此时，只需要统计频率作为两种概率的估计即可。

定义A，B，π并进行行规范化

定义状态转移矩阵$\mathbf{A}$：在人民日报数据集中，每个状态转移到另一种状态的频数。

观测矩阵$\mathbf{B}$：每个字作为每种状态的频数

初始状态矩阵π：句首是每种状态的频数

统计完频数之后，对矩阵和数组进行行规范化，使其变成概率。

对概率求对数

需要注意的是，由于对数在0处没有定义，需要将$\log 0$替换为一个非常小的负数。

Viterbi 算法

输入未分词的文本，得到Viterbi算法的T。对t进行遍历，回溯后得到状态序列。

$$
\begin{aligned}
&1.\text{Initialize } \delta_j(1)=\pi_j b_{jv(1)} \ and \ \psi_j(1)=0 \ (1\le j\le c)\\
&2.\text{For } t=2 \ to \ T\\
&3.\qquad \text{For } j=1 \ to \ c\\
&4.\qquad\qquad \delta_j(t)=\left[\max_{1\le i\le c}\delta_i(t-1)a_{ij}\right]b_{jv(t)};\ \psi_j(t)=\arg\max_{1\le i\le c}\delta_i(t-1)a_{ij}\\
&5.\qquad \text{End}\\
&6.\text{End}\\
&7.\text{Decode } \omega^*(T)=\arg\max_{1\le j\le c}\delta_j(T)\\
&8.\text{Decode } \omega^*(t)=\psi_{\omega^*(t+1)}(t+1)(1\le t\le T-1) \text{ with path backtracking (路径回溯)}
\end{aligned}
$$

分词

根据Viterbi算法返回的状态序列，判定字词的类别。设定如下：

单字词S状态为3，词的开始B状态为0，词的中间M状态为1，词的结束E状态为2。

返回分词的结果。

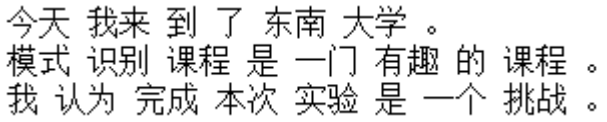
3.实验结果与分析

实验结果

实验中，给定没有分词的句子如下：

> 今天我来到了东南大学。
> 模式识别课程是一门有趣的课程。
> 我认为完成本次实验是一个挑战。

将输出的单词列表用空格连接，结果如下：



分词效果非常不错！除了"我来"这一分词结果在该语境下不太符合，其他都相对正确。

在尝试了其他的一些词句，观察更多的输入-输出后，发现该模型在大部分现代汉语情景中表现都不错，但它倾向于得到单字词和双字词，尤其是双字词，很少会得到多字词，所以在部分语句中的效果不是很好。

多字词数量较少的原因

隐马尔科夫模型采用的是一阶马尔可夫假设，它只考虑到相邻（尤其是前一个）字的关系，它不能考虑到不相邻的字之间的关系。

语料库中本身含有比较少的多字词，导致在学习状态转移概率的时候，B-M，M-M的概率比较小。

减少精度损失

实际计算中，数值的精度是有限的，由于涉及到多个概率的连续乘法，会产生精度下溢的现象。因此，对所有的概率取对数，后面在进行概率相乘时，变相乘为相加。

时间复杂度分析

使用Viterbi算法运用了动态规划的思想，使得原本的计算复杂度从$O(c^T \cdot T)$降低为$O(c^2 \cdot T)$。在使用对数后，还能有效降低计算复杂度。

4.代码附录

```
import numpy as np

#将频数转化概率，在转为log形式，以避免精度下溢
def log_normalize(array):
    total = array.sum()
    array /= total  #规范化为概率

    result = np.empty_like(array)
    for i in range(len(array)):
        if array[i] == 0:
            result[i] = -3.14e+100  #用极小数值代替log(0)
        else:
            result[i] = np.log(array[i])
    return result

#将一行用BMES编码
def encode(text):
    words = text.split()  #数据集用的是空格分词
    state_list = []
    sentence = ''.join(words)  #连起来的句子
    for i, word in enumerate(words):
        if len(word) == 1:  #单字词
            state_list.append(3)  #3代表状态S
        else:  #多字词
            state_list.extend([0] + (len(word) - 2) * [1] + [2])  #0,1,2代表状态BME
    return list(zip(sentence, state_list))

#用于中文分词的隐马尔科夫模型
class HMM:

    #初始化theta:
        #Pi: initial state probability,
        #A: transition probability
        #B: emitting probability
    def __init__(self, params_path=None):
        if params_path:
            params = np.load(params_path)
            self.init_prob = params['init_prob']
            self.trans_prob = params['trans_prob']
            self.emit_prob = params['emit_prob']
        else:
            self.init_prob = np.zeros(4) #4:BMES4种状态
            self.trans_prob = np.zeros((4, 4))
            self.emit_prob = np.zeros((4, 65536)) #65536:确保sentence的长度不会超过emit_prob的长度
```

```
    #根据词频统计学习隐马尔可夫模型
        #:param file_path: utf-8 encoded Chinese separated text
    def train(self, file_path, save_to=None):
        with open(file_path, 'r', encoding='utf-8') as f:
            for line in f.readlines():
                encoding = encode(line)
                for i in range(len(encoding)):
                    char, state = encoding[i]
                    char_index = ord(char)
                    self.emit_prob[state, char_index] += 1  #统计频数，更新观测矩阵
                    if i == 0:  #句首
                        self.init_prob[state] += 1  #统计频数，更新初始概率
                    else:
                        prev_state = encoding[i - 1][1]
                        self.trans_prob[prev_state, state] += 1  #统计频数，更新状态转移矩阵

        #将频数规范化为频率，并取对数
        self.init_prob = log_normalize(self.init_prob)
        for i in range(self.trans_prob.shape[0]):
            self.trans_prob[i] = log_normalize(self.trans_prob[i])
            self.emit_prob[i] = log_normalize(self.emit_prob[i])

        #:param save_to: save model parameters to a file
        if save_to:
            np.savez(save_to, trans_prob=self.trans_prob, emit_prob=self.emit_prob,
init_prob=self.init_prob)

    #用Viterbi算法对一行文本进行解码
    def Viterbi(self, text):
        """
        :param text: 未分词的文本
        :return: 状态序列
        """
        length = len(text)
        delta = np.zeros((length, 4))
        psi = np.zeros((length, 4), dtype=int)

        #计算delta和psi
        #由于log，概率的乘法都变加法
        for t in range(length):
            char_index = ord(text[t])
            if t == 0:
                delta[t] = self.init_prob + self.emit_prob[:, char_index]  #delta_j(0) =
pi_j*b_j_v(0)
            else:
                for j in range(4):
                    temp = delta[t - 1] + self.trans_prob[:, j]  #delta_i(t-1)*a_ij
                    psi[t, j] = np.argmax(temp)
                    delta[t, j] = temp.max() + self.emit_prob[j, char_index]

        #开始回溯
```

```
        omega = np.zeros(length, dtype=int)
        omega[-1] = np.argmax(delta[-1])
        for t in range(length - 2, -1, -1):
            omega[t] = psi[t + 1, omega[t + 1]]

        return omega


    #根据状态序列得到一行文本的分词结果
       #:param text: 未分词的文本
    def split(self, text):
        text = text.strip()
        state_list = self.Viterbi(text)
        word_list = []
        for i in range(len(text)):
            if state_list[i] == 3:  #S，表示单字词
                word_list.append(text[i])
            elif state_list[i] == 0:  #B，表示一个词的开始
                word = text[i]
            elif state_list[i] == 1:  #M，表示一个词的中间
                word += text[i]
            else:  #E，表示一个词的结束
                word += text[i]
                word_list.append(word)

        #返回分词结果
        return word_list


def main():
    test_sentences = ["今天我来到了东南大学。",
                      "模式识别课程是一门有趣的课程。",
                      "我认为完成本次实验是一个挑战。"]

    try:
        split_model = HMM("hmm_params.npz")
    except:
        split_model = HMM()
        split_model.train("./RenMinData.txt_utf8", save_to="hmm_params.npz")
    for sent in test_sentences:
        words = split_model.split(sent)
        print(" ".join(words))#使用空格进行分词


if __name__ == "__main__":
    main()
```

心得体会

布置到完成经历了整整两周时间，但实际上花费在代码上的时间就要超过一周。在辛苦完成代码部分的编写、调试之后，撰写报告时才发现还有一些可视化的需求需要额外添加。整个过程不仅是技巧上的模联，更是是耐心的考验。

印象最深刻的是KNN分类任务。在欧氏距离、切比雪夫距离和曼哈顿距离三种距离度量下，运用归一化得到的结果令人咂舌，让人怀疑一定是哪里出了问题：k=1时正确率是最小的，与去掉归一化后的结果大相径庭。对这个问题百思不得其解，最终将原因归结到数据集身上。其次是进行马氏距离的训练时，第一次感受到了计算复杂带来的煎熬。从未在非神经网络的模型上尝试过训练四五个小时得到结果。这使得在调试马氏距离时需要格外小心，先对小规模的数据验证了代码的可行性，再将代码交给服务器来处理。

这次实验有痛苦，收获也非常之巨大。

首先是数据读写、模型保存这些操作可谓开始熟悉起来了。其次是脱离现有的库进行from scracth的编程，对于面向对象编程的理解更加深刻了。再者是在编程中复习了算法，更好地掌握课内只是。还有就是LaTeX的书写也得到了巩固，每一个公式的数学表达都力求做到严谨不出错。当然最重要的是提升了学习能力。写代码似乎不是闭门造车的事情，不仅仅需要查看文档和博客来研究数据类型、函数的参数等等，还需要参考和借鉴许多优秀的方法，学习到巧妙的对象设计和函数实现，让人大开眼界。即使这些锻炼对于工程上帮助不大，但提升查阅、搜集、消化知识的能力显得非常之珍贵。

最后感慨写文档繁琐的同时，又感谢写代码时随手写下注释的习惯。这会对我之后的编程有十分重要的提醒作用。