实验三 逻辑回归与 BP 神经网络分类器

一、实验目的及要求   
 1、掌握 Python 安装、编程环境搭建的法。

2、熟练掌握逻辑归回分类器的工作原理。

3、掌握 BP 神经网络分类器的基本工作原理。

4、理解 BP 神经网络激活函数以及学习率等参数的调节方法

二、预习要求   
 阅读本实验例程部分，实现基本的逻辑回归以及 BP 神经网络分类方法，以便够

充分利用实验时间编程调试。

三、实验设备   
 硬件：PC 机。

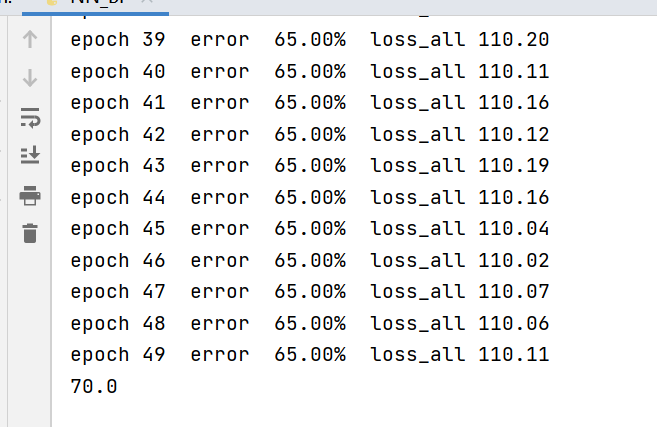
软件：Python 及相关集成开发环境。

四、实验内容   
利用利用 Python 编程实现基于决策树分类算法，并将训练得到决策树绘制来。

具体要求如下：

1. 编写逻辑回归分类器的代码，并利用逻辑回归分类器对鸢尾花数据集进行分类

import numpy as np  
from random import shuffle  
  
  
def softmax(x):  
 return np.exp(x) / np.sum(np.exp(x), axis=1, keepdims=True)  
  
  
def cross\_entropy\_loss(y\_hat, y):  
 return -np.mean(np.log(y\_hat[np.arange(len(y\_hat)), y]))  
  
  
def convert\_one\_hot(labs, num\_classes):  
 one\_hot\_label = np.eye(num\_classes)[labs]  
 return one\_hot\_label  
  
  
*# datas NxD  
# labs Nx1  
# w Dx1*def weight\_update(datas, labs, w, b, alpha=0.01, num\_classes=3):  
 z = np.dot(datas, w) + b *# Nx1* h = softmax(z) *# Nx1* Error = h - convert\_one\_hot(labs, num\_classes) *# Nx1  
 # Error = cross\_entropy\_loss(h, labs)  
 # print(Error)* w = w - alpha \* np.dot(datas.T, Error)  
 b = b - alpha \* np.sum(Error, axis=0, keepdims=True)  
 return w, b  
  
  
*# 随机梯度下降*def train\_LR\_batch(datas, labs, batchsize, n\_epoch=2, alpha=0.005, num\_classes=3):  
 N, D = np.shape(datas)  
 *# weight 初始化* w = np.random.rand(D, num\_classes) *# Dx1* b = np.zeros((1, num\_classes))  
 N\_batch = N // batchsize  
 for i in range(n\_epoch):  
 *# 数据打乱* rand\_index = np.random.permutation(N).tolist()  
 *# 每个batch 更新一下weight* for j in range(N\_batch):  
 index = rand\_index[j \* batchsize:(j + 1) \* batchsize]  
 batch\_datas = datas[index]  
 batch\_labs = labs[index]  
 w, b = weight\_update(batch\_datas, batch\_labs, w, b, alpha, num\_classes)  
  
 error = test\_accuracy(datas, labs, w, b, num\_classes)  
 print("epoch %d error %.2f%%" % (i, error \* 100))  
 return w, b  
  
  
def test\_accuracy(datas, labs, w, b, num\_classes):  
 N, D = np.shape(datas)  
 z = np.dot(datas, w) + b *# Nx1* h = softmax(z) *# Nx1* lab\_det = (h > 0.5).astype(np.float64)  
 one\_hot = convert\_one\_hot(labs, num\_classes)  
 temp = np.abs(one\_hot - lab\_det)  
 error\_num = 0  
 for i in temp:  
 if sum(i) == 0:  
 pass  
 else:  
 error\_num = error\_num + 1  
 error\_rate = error\_num / N  
 return error\_rate  
  
  
def dataSplit(path, ratio):  
 with open(path, 'r') as f:  
 lines = f.read().splitlines()  
  
 shuffle(lines)  
  
 train\_data = lines[: int(len(lines) \* ratio)]  
 test\_data = lines[int(len(lines) \* ratio):]  
  
 return train\_data, test\_data  
  
  
def convert\_label(label):  
 for i in range(len(label)):  
 if label[i] == 'Iris-setosa':  
 label[i] = 0  
 elif label[i] == 'Iris-versicolor':  
 label[i] = 1  
 else:  
 label[i] = 2  
  
 return label  
  
  
def create\_dataset\_iris(data\_all):  
 label = [line.split(",")[-1] for line in data\_all]  
 label = convert\_label(label)  
 label = np.array(label).astype(np.int64)  
  
 data = [line.split(",")[:-1] for line in data\_all]  
 data = np.array(data).astype(np.float64)  
  
 return data, label  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
 file\_data = "iris.data"  
  
 *# 数据读取* train\_datas, test\_datas = dataSplit(file\_data, 0.7)  
  
 *# 获取数据个数 以及维度* train\_data, train\_label = create\_dataset\_iris(train\_datas)  
 test\_data, test\_label = create\_dataset\_iris(test\_datas)  
  
 *# 随机梯度下降* weights, bias = train\_LR\_batch(train\_data, train\_label, batchsize=8, n\_epoch=1000, alpha=0.0001, num\_classes=3)  
 print(f'The final weights:\n{weights}')  
 print(f'The final bias:\n{bias}')  
  
 acc = 1 - test\_accuracy(test\_data, test\_label, weights, bias, num\_classes=3)  
 print('Acc:{:.2f}%'.format(acc \* 100))

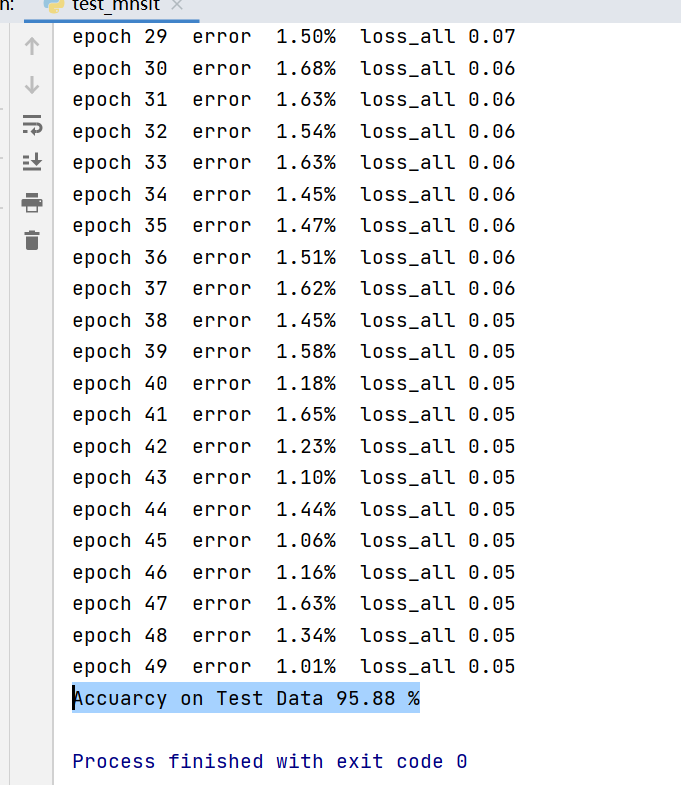


（2）自行设计测试方案，尝试调节学习步长，学习轮次（epoch）以及 batchsize 的大

小，观测其对训练速度和检测精度的影响

1. 编程实现 BP 神经网络分类器，并利用其对 MNIST 数据集进行分类

import numpy as np  
  
  
def sigmod(z):  
 h = 1. / (1 + np.exp(-z))  
 return h  
  
  
def de\_sigmoid(z, h):  
 return h \* (1 - h)  
  
  
def relu(z):  
 h = np.maximum(z, 0)  
 return h  
  
  
def de\_relu(z, h):  
 z[z <= 0] = 0  
 z[z > 0] = 1.0  
 return z  
  
  
def no\_active(z):  
 h = z  
 return h  
  
  
def de\_no\_active(z, h):  
 return np.ones(h.shape)  
  
  
*# o Nxc  
# lab Nxc*def loss\_L2(o, lab):  
 diff = lab - o  
 sqrDiff = diff \*\* 2  
 return 0.5 \* np.sum(sqrDiff)  
  
  
def de\_loss\_L2(o, lab):  
 return o - lab  
  
  
def loss\_CE(o, lab):  
 p = np.exp(o) / np.sum(np.exp(o), axis=1, keepdims=True)  
 loss\_ce = np.sum(-lab \* np.log(p))  
 return loss\_ce  
  
  
def de\_loss\_CE(o, lab):  
 p = np.exp(o) / np.sum(np.exp(o), axis=1, keepdims=True)  
 return p - lab  
  
  
*# dim\_in:输入特征的维度  
# list\_num\_hidden： 每层输出节点的数目  
# list\_act\_funs： 每层的激活函数  
# list\_de\_act\_funs: 反向传播时的函数*def bulid\_net(dim\_in, list\_num\_hidden,  
 list\_act\_funs, list\_de\_act\_funs):  
 layers = []  
  
 *# 逐层的进行网络构建* for i in range(len(list\_num\_hidden)):  
 layer = {}  
  
 *# 定义每一层的权重* if i == 0:  
 *# layer["w"]= 0.2\*np.random.randn(dim\_in,list\_num\_hidden[i])-0.1 # 用sigmoid激活函数* layer["w"] = 0.01 \* np.random.randn(dim\_in, list\_num\_hidden[i]) *# 用relu 激活函数* else:  
 *# layer["w"]= 0.2\*np.random.randn(list\_num\_hidden[i-1],list\_num\_hidden[i])-0.1 # 用sigmoid激活函数* layer["w"] = 0.01 \* np.random.randn(list\_num\_hidden[i - 1], list\_num\_hidden[i]) *# 用relu 激活函数  
  
 # 定义每一层的偏置* layer["b"] = 0.1 \* np.ones([1, list\_num\_hidden[i]])  
 layer["act\_fun"] = list\_act\_funs[i]  
 layer["de\_act\_fun"] = list\_de\_act\_funs[i]  
 layers.append(layer)  
  
 return layers  
  
  
*# 返回每一层的输入  
# 与最后一层的输出*def fead\_forward(datas, layers):  
 input\_layers = []  
 input\_acfun = []  
 for i in range(len(layers)):  
 layer = layers[i]  
 if i == 0:  
 inputs = datas  
 z = np.dot(inputs, layer["w"]) + layer["b"]  
 h = layer['act\_fun'](z)  
 input\_layers.append(inputs)  
 input\_acfun.append(z)  
 else:  
 inputs = h  
 z = np.dot(inputs, layer["w"]) + layer["b"]  
 h = layer['act\_fun'](z)  
 input\_layers.append(inputs)  
 input\_acfun.append(z)  
 return input\_layers, input\_acfun, h  
  
  
*# 进行参数更新更新*def updata\_wb(datas, labs, layers, loss\_fun, de\_loss\_fun, alpha=0.01):  
 N, D = np.shape(datas)  
 *# 进行前馈操作* inputs, input\_acfun, output = fead\_forward(datas, layers)  
 *# 计算 loss* loss = loss\_fun(output, labs)  
 *# 从后向前计算* deltas0 = de\_loss\_fun(output, labs)  
 *# 从后向前计算误差* deltas = []  
 for i in range(len(layers)):  
 index = -i - 1  
 if i == 0:  
 h = output  
 z = input\_acfun[index]  
 delta = deltas0 \* layers[index]["de\_act\_fun"](z, h)  
 else:  
 h = inputs[index + 1]  
 z = input\_acfun[index]  
 *# print(layers[index]["de\_act\_fun"](z,h)[1])* delta = np.dot(delta, layers[index + 1]["w"].T) \* layers[index]["de\_act\_fun"](z, h)  
  
 deltas.insert(0, delta)  
  
 *# 利用误差 对每一层的权重进行修成* for i in range(len(layers)):  
 *# 计算 dw 与 db* dw = np.dot(inputs[i].T, deltas[i])  
 db = np.sum(deltas[i], axis=0, keepdims=True)  
 *# 梯度下降* layers[i]["w"] = layers[i]["w"] - alpha \* dw  
 layers[i]["b"] = layers[i]["b"] - alpha \* db  
  
 return layers, loss  
  
  
def test\_accuracy(datas, labs\_true, layers):  
 \_, \_, output = fead\_forward(datas, layers)  
 lab\_det = np.argmax(output, axis=1)  
 labs\_true = np.argmax(labs\_true, axis=1)  
 N\_error = np.where(np.abs(labs\_true - lab\_det) > 0)[0].shape[0]  
  
 error\_rate = N\_error / np.shape(datas)[0]  
 return error\_rate  
  
  
def load\_dataset\_iris(file\_data, N\_train):  
 *# 数据读取* datas = np.loadtxt(file\_data, dtype=np.float64, delimiter=',', usecols=(0, 1, 2, 3))  
 labs = np.loadtxt(file\_data, dtype=str, delimiter=',', usecols=(4))  
 N, D = np.shape(datas)  
 N\_test = N - N\_train  
 unqiue\_labs = np.unique(labs).tolist()  
  
 dic\_str2index = {}  
 dic\_index2str = {}  
 for i in range(len(unqiue\_labs)):  
 lab\_str = unqiue\_labs[i]  
 dic\_str2index[lab\_str] = i  
 dic\_index2str[i] = lab\_str  
  
 labs\_onehot = np.zeros([N, len(unqiue\_labs)])  
 for i in range(N):  
 labs\_onehot[i, dic\_str2index[labs[i]]] = 1  
  
 perm = np.random.permutation(N)  
 index\_train = perm[:N\_train]  
 index\_test = perm[N\_train:]  
  
 data\_train = datas[index\_train, :]  
 lab\_train\_onehot = labs\_onehot[index\_train, :]  
  
 data\_test = datas[index\_test, :]  
 lab\_test\_onehot = labs\_onehot[index\_test]  
  
 return data\_train, lab\_train\_onehot, data\_test, lab\_test\_onehot, dic\_index2str  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
 file\_data = 'iris.data'  
  
 data\_train, lab\_train\_onehot, data\_test, lab\_test\_onehot, dic\_index2str = load\_dataset\_iris(file\_data, 100)  
  
 N, dim\_in = np.shape(data\_train)  
 *# 定义网络结构* list\_num\_hidden = [10, 5, 3]  
 list\_act\_funs = [relu, relu, no\_active]  
 list\_de\_act\_funs = [de\_relu, de\_relu, de\_no\_active]  
  
 *# 定义损失函数* loss\_fun = loss\_CE  
 de\_loss\_fun = de\_loss\_CE  
  
 *# loss\_fun = loss\_L2  
 # de\_loss\_fun=de\_loss\_L2* layers = bulid\_net(dim\_in, list\_num\_hidden,  
 list\_act\_funs, list\_de\_act\_funs)  
  
 *# 进行训练* n\_epoch = 50  
 batchsize = 4  
 N\_batch = N // batchsize  
 for i in range(n\_epoch):  
 *# 数据打乱* rand\_index = np.random.permutation(N).tolist()  
 *# 每个batch 更新一下weight* loss\_sum = 0  
 for j in range(N\_batch):  
 index = rand\_index[j \* batchsize:(j + 1) \* batchsize]  
 batch\_datas = data\_train[index]  
 batch\_labs = lab\_train\_onehot[index]  
 layers, loss = updata\_wb(batch\_datas, batch\_labs, layers, loss\_fun, de\_loss\_fun, alpha=0.01)  
 loss\_sum = loss\_sum + loss  
  
 error = test\_accuracy(data\_train, lab\_train\_onehot, layers)  
 print("epoch %d error %.2f%% loss\_all %.2f" % (i, error \* 100, loss\_sum))  
  
 *# 进行测试* error = test\_accuracy(data\_test, lab\_test\_onehot, layers)  
 print(error \* 100)



（4）自行设计测试方案，调节神经网络的层数、每层的节点数、激活函数以及损失函

数，记录调节后对模型训练速度以及测试精度的影响。

import numpy as np  
from NN\_BP import \*  
  
  
def load\_mnist(file\_data, file\_lab):  
 *# 加载训练数据* data = np.load(file\_data)  
 lab = np.load(file\_lab)  
 N, D = np.shape(data)  
  
 *# 构造 one-hot 标签* lab\_onehot = np.zeros([N, 10])  
 for i in range(N):  
 id = int(lab[i, 0])  
 lab\_onehot[i, id] = 1  
 data = (data.astype(np.float64) / 255.0)  
 return data, lab\_onehot  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
 *# 加载训练数据* train\_data, train\_lab\_onehot = load\_mnist("train\_data.npy", "train\_lab.npy")  
 N, D = np.shape(train\_data)  
  
 *# 搭建网络  
 # 定义网络结构* list\_num\_hidden = [30, 5, 10]  
  
 *# list\_act\_funs =[sigmod,sigmod,no\_active]  
 # list\_de\_act\_funs=[de\_sigmoid,de\_sigmoid,de\_no\_active]  
  
 # # 定义损失函数  
 # loss\_fun = loss\_L2  
 # de\_loss\_fun=de\_loss\_L2* list\_act\_funs = [relu, relu, no\_active]  
 list\_de\_act\_funs = [de\_relu, de\_relu, de\_no\_active]  
 *# 定义损失函数* loss\_fun = loss\_CE  
 de\_loss\_fun = de\_loss\_CE  
  
 layers = bulid\_net(D, list\_num\_hidden,  
 list\_act\_funs, list\_de\_act\_funs)  
  
 *# 进行训练* n\_epoch = 50  
 batchsize = 20  
 N\_batch = N // batchsize  
 for i in range(n\_epoch):  
 *# 数据打乱* rand\_index = np.random.permutation(N).tolist()  
 *# 每个batch 更新一下weight* loss\_sum = 0  
 for j in range(N\_batch):  
 index = rand\_index[j \* batchsize:(j + 1) \* batchsize]  
 batch\_datas = train\_data[index]  
 batch\_labs = train\_lab\_onehot[index]  
 layers, loss = updata\_wb(batch\_datas, batch\_labs, layers, loss\_fun, de\_loss\_fun, alpha=0.001)  
 *# print("epoch %d batch %d loss %.2f"%(i,j,loss/batchsize))* loss\_sum = loss\_sum + loss  
  
 error = test\_accuracy(train\_data, train\_lab\_onehot, layers)  
 print("epoch %d error %.2f%% loss\_all %.2f" % (i, error \* 100, loss\_sum / (N\_batch \* batchsize)))  
  
 np.save("model.npy", layers)  
  
 *# 加载测试数据* test\_data, test\_lab\_onehot = load\_mnist("test\_data.npy", "test\_lab.npy")  
 layers = np.load("model.npy", allow\_pickle=True)  
  
 error = test\_accuracy(test\_data, test\_lab\_onehot, layers)  
 print("Accuarcy on Test Data %.2f %%" % ((1 - error) \* 100))

