逻辑回归模型运用实例

姓名 王端初 学号 201904195031

1. 逻辑回归原理
2. 数据集分析&主要代码和结果图

import pandas as *pd*import numpy as *np*import matplotlib.pyplot as *plt*import torch  
import torch.utils.data as *Data*import time  
import tqdm  
import os  
import seaborn as *sns*os.environ["KMP\_DUPLICATE\_LIB\_OK"]="TRUE"

导入需要用的python库，pandas和numpy用于数据的读取与分析，matplotlib和seaborn

用于对数据进行可视化，选择pytorch框架搭建逻辑回归模型，time和tqdm对训练过程进行可视化。

# 定义超参数  
*epoch\_num*=100  
*device* = torch.device("cuda:0" if (torch.cuda.is\_available()) else "cpu")  
*Learning\_Rate*=1e-3  
*LOSS*=[]  
*ACC*=[]  
*Recall*=[]  
*Precision*=[]  
*f1\_score*=[]

定义超参数，训练轮次为50，有GPU时使用CUDA没GPU下使用CPU进行计算，优化器的学习率定义为0.001，并且为后面的画图定义存储的列表。

# 读取数据  
*data*=pd.read\_csv("diabetes.csv")  
# 分析数据相关性  
plt.figure(figsize=(10,10))  
sns.heatmap(*data*.corr(),annot=True)  
plt.savefig('corr.png', bbox\_inches='tight')  
plt.show()  
plt.close()

读取数据，可知数据大小为（768，9），最后一列Outcome为标签，分析其他八个特征和标签的相关性，用热力图表示。

热力图它是表示两个数据之间的相关性，数值范围是-1到1之间，大于0表示两个数据是正相关的，小于0表示两个数据是负相关的，等于0就是不相关。我们可以看到有一条对角线上的数值都是1，两个数据如果是相同的两个数据，那么他们的相关系数就是1。并且这个相关系数矩阵是对称的。我们可以观察到一些比较明显的特征，比如Age年龄和Pregnancies怀孕次数有比较强的相关性，也就是说年龄越大怀孕次数就越多，或者怀孕次数越多年龄就越大，这应该是比较合理的一个情况。糖尿病的标签Outcome和Glucose葡萄糖测试值正相关系数比较大，也就是说葡萄糖测试值比较高的话，那么就很可能患有糖尿病。

########################################################

0.8-1.0 极强相关

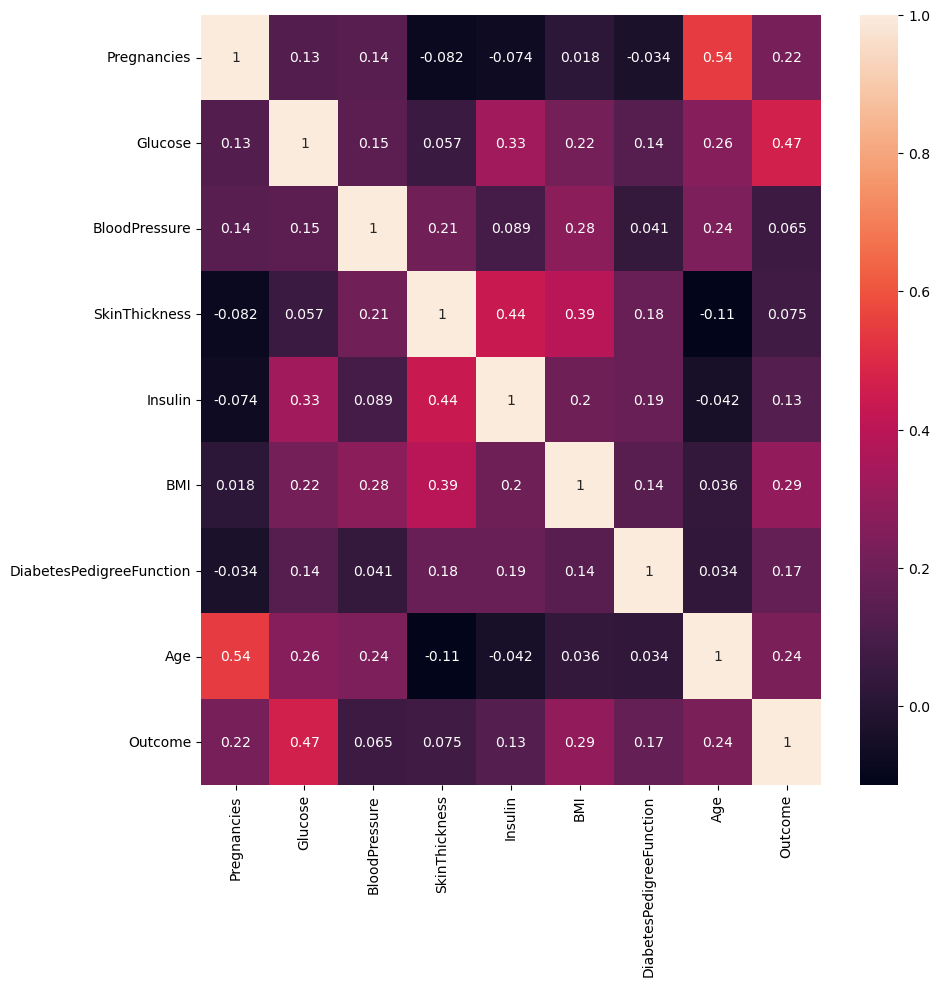
0.6-0.8 强相关

0.4-0.6 中等程度相关

0.2-0.4 弱相关

0.0-0.2 极弱相关或无相关

########################################################



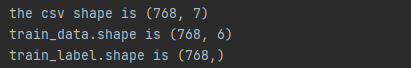
据此我们可以看出BloodPressure、SkinThickness这两个特征和我们数据的标签相关性不大，可能会污染数据，所以在此我们将这两个数据移除，构成新的数据。

# 转换为numpy  
*data*=pd.read\_csv("diabetes.csv",usecols=['Pregnancies','Glucose','Insulin','BMI','DiabetesPedigreeFunction','Age','Outcome'])  
*np\_data*=np.array(*data*)  
  
# 数据归一化  
for *i* in range(6):  
 *r*=*np\_data*[:,*i*]  
 *np\_data*[:,*i*]=(*r* - np.min(*r*)) / (np.max(*r*) - np.min(*r*))

## 重新读取数据，对数据进行归一化。归一化的第一个好处是加快迭代速度，第二好处是提高[精度](https://so.csdn.net/so/search?q=%E7%B2%BE%E5%BA%A6&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_38313518/article/details/_blank)，这在涉及到一些距离计算的算法时效果显著。由于多维度的量纲不同，这就会造成精度的损失。所以归一化很有必要，他可以让各个特征对结果做出的贡献相同。这里我选择Min-max normalization的方法进行归一化。

# 打印数据  
print(*data*)  
print("the csv shape is {}".format(*np\_data*.shape))  
# 分为data/label  
*train\_data*=*np\_data*[:,:6]  
*train\_label*=*np\_data*[:,6]  
# 查看数据大小  
print("train\_data.shape is {}".format(*train\_data*.shape))  
print("train\_label.shape is {}".format(*train\_label*.shape))

打印并查看数据，将数据分为data和label，查看数据维度。可以看



# np->tensor  
*train\_data\_tensor*=torch.from\_numpy(*train\_data*).to(*device*).float()  
*train\_label\_tensor*=torch.from\_numpy(*train\_label*).to(*device*).long()  
print("the tensor of data is size of {}".format(*train\_data\_tensor*.shape))  
print("the tensor of label is size of {}".format(*train\_label\_tensor*.shape))

因为pytorch计算单位是张量tensor，numpy的单位是矩阵，所以将numpy的矩阵转为tensor，查看数据维度。



# 划分训练集/验证集  
*torch\_dataset* = Data.TensorDataset(*train\_data\_tensor*,*train\_label\_tensor*)  
*train\_size* = int(len(*torch\_dataset*) \* 0.8)  
*val\_size* = len(*torch\_dataset*) - *train\_size  
train\_dataset*, *val\_dataset* = torch.utils.data.random\_split(*torch\_dataset*, [*train\_size*, *val\_size*])  
*train\_loader*=Data.DataLoader(*train\_dataset*,batch\_size=128,shuffle=True)  
*val\_loader*=Data.DataLoader(*val\_dataset*,batch\_size=128,shuffle=True)  
# 查看数据集  
print("train\_loader length is {}".format(len(*train\_loader*)))  
print("val\_loader length is {}".format(len(*val\_loader*)))

于模型来说，其在训练集上面的误差我们称之为 训练误差 或者 经验误差，而在测试集上的误差称之为 测试误差。因为测试集是用来测试学习器对新样本的学习能力，因此我们可以把测试误差作为 泛化误差 的近似（泛化误差：在新样本上的误差）。

对于我们来说，我们更关心的是模型对于新样本的学习能力，即我们希望通过对已有样本的学习，尽可能的将所有潜在样本的普遍规律学到手，而如果模型对训练样本学的太好，则有可能把训练样本自身所具有的一些特点当做所有潜在样本的普遍特点，这时候我们就会出现 过拟合 的问题。

因此在这里我们通常将已有的数据集划分为训练集和测试集两部分，其中训练集用来训练模型，而测试集则是用来评估模型对于新样本的判别能力。

对于数据集的划分，我们通常要保证满足以下两个条件：

1. 训练集和测试集的分布要与样本真实分布一致，即训练集和测试集都要保证是从样本真实分布中独立同分布采样而得；
2. 训练集和测试集要互斥

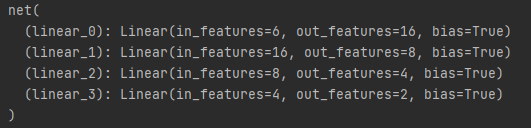
所以选择随机8：2划分，一个批次为128。



# 定义网络  
class net(torch.nn.Module):  
 def \_\_init\_\_(self):  
 super(net, self).\_\_init\_\_()  
 self.linear\_0 = torch.nn.Linear(6, 16)  
 self.linear\_1 = torch.nn.Linear(16, 8)  
 self.linear\_2 = torch.nn.Linear(8, 4)  
 self.linear\_3 = torch.nn.Linear(4, 2)  
 def forward(self,*x*):  
 *linear\_0*=self.linear\_0(*x*)  
 *linear\_1* = self.linear\_1(*linear\_0*)  
 *linear\_2* = self.linear\_2(*linear\_1*)  
 *linear\_3* = self.linear\_3(*linear\_2*)  
 return *linear\_3*

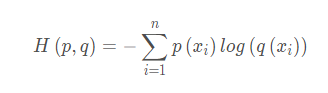
定义自己的网络，继承网络的基类，定义4个线性模型用于分类。Forward函数定义前向传播。

输入维度为6，输出维度为2，用于分类。



# 定义网络\损失函数\优化器  
*Net*=net().to(*device*)  
*criterion*=torch.nn.CrossEntropyLoss()  
*opt*=torch.optim.Adam(*Net*.parameters(),lr=*Learning\_Rate*)

定义自己的网络，选择交叉熵作为损失函数，选择Adam做梯度下降，学习率为开始定义的0.001。

交叉熵的公式为

交叉熵能够衡量同一个随机变量中的两个不同概率分布的差异程度，在机器学习中就表示为真实概率分布与预测概率分布之间的差异。交叉熵的值越小，模型预测效果就越好。交叉熵在分类问题中常常与[softmax](https://so.csdn.net/so/search?q=softmax&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/b1055077005/article/details/_blank)是标配，softmax将输出的结果进行处理，使其多个分类的预测值和为1，再通过交叉熵来计算损失。

# 训练  
for *epoch* in range(*epoch\_num*):  
 *start*=time.time()  
 *running\_loss* = 0.0  
 for *x*,*y* in tqdm.tqdm(iter(*train\_loader*)):  
 *pre*=*Net*(*x*)  
 *loss*=*criterion*(*pre*,*y*)  
 *running\_loss*=*running\_loss*+*loss*/128  
 *opt*.zero\_grad()  
 *loss*.backward()  
 *opt*.step()  
 *LOSS*.append(*running\_loss*)  
 *end* = time.time()  
 print("train {} epoch ,total loss is {},time use {}s".format(*epoch*+1,*running\_loss*,*end*-*start*))

训练的过程，训练100个轮次，并且计算每个轮次所用的时间。再代码中可以看到，

更新梯度

反向传播

梯度归零

计算损失

前向传播

with torch.no\_grad():  
 *total*=0  
 *correct*=0  
 *TP*=0  
 *FP*=0  
 *TN*=0  
 *FN*=0  
 *start* = time.time()  
 *running\_loss* = 0.0  
 for *x*, *y* in tqdm.tqdm(iter(*val\_loader*)):  
 *pre* = *Net*(*x*)  
 *\_*, *predicted* = torch.max(*pre*, -1)  
 *total* += *y*.size(0)  
 *correct* += (*predicted* == *y*.view(-1)).sum().item()  
 *y\_true* = *y*.detach().cpu().numpy()  
 *y\_pred* = *predicted*.detach().cpu().numpy()  
 *TP* += np.sum(np.logical\_and(np.equal(*y\_true*, 1), np.equal(*y\_pred*, 1)))  
 *FP* += np.sum(np.logical\_and(np.equal(*y\_true*, 0), np.equal(*y\_pred*, 1)))  
 *TN* += np.sum(np.logical\_and(np.equal(*y\_true*, 1), np.equal(*y\_pred*, 0)))  
 *FN* += np.sum(np.logical\_and(np.equal(*y\_true*, 0), np.equal(*y\_pred*, 0)))  
 *precision* = *TP* / (*TP* + *FP*)  
 *recall* = *TP* / (*TP* + *FN*)  
 *F1\_Score* = 2 \* *precision* \* *recall* / (*precision* + *recall*)  
 *acc*=*correct*/*total  
 ACC*.append(*acc*)  
 *f1\_score*.append(*F1\_Score*)  
 *Recall*.append(*recall*)  
 *Precision*.append(*precision*)  
 *end* = time.time()  
print("val {} epoch ,total acc is {}%,time use {}s".format(*epoch* + 1, *acc*, *end* - *start*))

验证集测试，计算TP/FP/FN/TN。

TP(True Positive)：将正类预测为正类（的数目），真实为0，预测也为0;

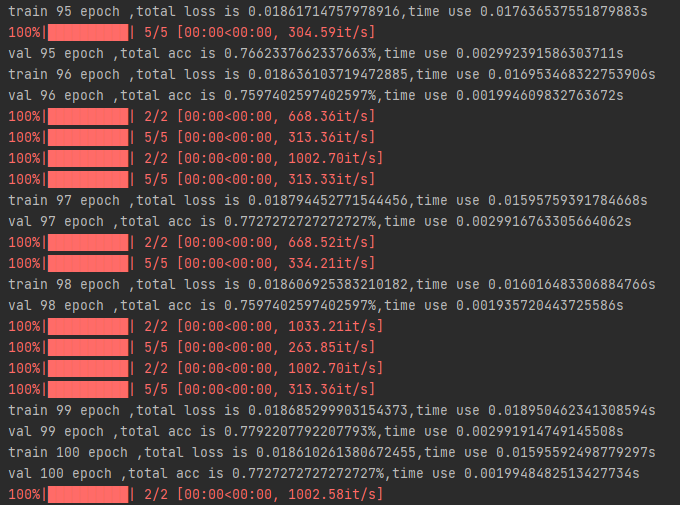
FN(False Negative)：将正类预测为负类（的数目），真实为0，预测为1;

FP(False Positive)：将负类预测为正类（的数目）， 真实为1，预测为0;

TN(True Negative)：将负类预测为负类（的数目），真实为1，预测也为1。

精确率（precision）：TP/(TP+FP)  
召回率（recall）：TP/(TP+FN)

F1\_score : 2\* precision \* recall / (recall+precision)



取最后五个轮次数据进行可视化。

# 画图  
plt.figure(1)  
plt.plot(range(*epoch\_num*), *ACC*, '.-b', label='Acc')  
plt.xlabel('Epoch')  
plt.ylabel('Acc')  
plt.legend()  
plt.savefig('Acc.png', bbox\_inches='tight')  
plt.show()  
plt.close()  
  
plt.figure(2)  
plt.plot(range(*epoch\_num*), *LOSS*, '.-r', label='Loss')  
plt.xlabel('Epoch')  
plt.ylabel('Loss')  
plt.legend()  
plt.savefig('Loss.png', bbox\_inches='tight')  
plt.show()  
plt.close()  
  
plt.figure(3)  
plt.plot(range(*epoch\_num*), *Recall*, '.-b', label='Recall')  
plt.xlabel('Epoch')  
plt.ylabel('Recall')  
plt.legend()  
plt.savefig('Recall.png', bbox\_inches='tight')  
plt.show()  
plt.close()  
  
plt.figure(4)  
plt.plot(range(*epoch\_num*), *Precision*, '.-y', label='Precision')  
plt.xlabel('Epoch')  
plt.ylabel('Precision')  
plt.legend()  
plt.savefig('Precision.png', bbox\_inches='tight')  
plt.show()  
plt.close()  
  
plt.figure(5)  
plt.plot(range(*epoch\_num*), *f1\_score*, '.-g', label='f1\_score')  
plt.xlabel('Epoch')  
plt.ylabel('f1\_score')  
plt.legend()  
plt.savefig('f1\_score.png', bbox\_inches='tight')  
plt.show()  
plt.close()

画图代码，查看最后的结果图片。

