西北工业大学
NORTHWESTERN POLYTECHNICAL UNIVERSITY

模式识别课程报告

多种疾病预测
基于血液样本的疾病预测数据集

完成人：王浩宇 2022302098

2024 年 5 月 23 日

一、背景与意义

在今天的医疗环境中，一种更有效、及时的诊断手段显得尤为重要。而血液，作为体内循环的重要媒介，包含了众多有关人体状况的信息。像是各种生物化学指标、血细胞的数值和型态、还有各种微生物的存在等等。通过对这些信息进行深入挖掘和分析，我们有可能创建出一个准确的预测模型，以便在疾病初期就发现潜在的问题

目前，血液测试已经成为医学诊断中无法或缺的一部分，但是，传统的血液测试依赖于专业的医生对测试结果进行人工解读。在这种情况下，医生可能会因为过于关注某项特定的指标，而忽视了其他可能的数据关联。

机器学习作为一种强大的模式识别工具，正被广泛应用在许多领域，包括医疗健康。这些技术可以帮助我们发现在大量数据中隐藏的模式，这些模式在揭示着某种我们之前未曾注意到的规律。

我选择了**基于血液样本的疾病预测**作为大作业的主题，构建一个机器学习模型，通过分析血液测试数据预测疾病的发展。这样，我们可以早期预测疾病，尽早进行干预和治疗，从而在一定程度上提高治疗的成功率。

二、算法

1.贝叶斯分类器

（1）朴素贝叶斯

朴素贝叶斯（Naive Bayes）是基于贝叶斯定理与特征独立假设的分类方法。使用朴素贝叶斯方法时，首先基于训练数据，基于特征条件独立假设学习输入与输出的联合概率分布；随后对于给定的 X，利用贝叶斯定理求解后验概率最大的输出标签。

朴素贝叶斯本质属于生成模型，学习生成数据的机制，也就是联合概率分布。

（2）半朴素贝叶斯

半朴素贝叶斯是适当考虑一部分属性之间的相互依赖信息，其中“独依赖估计”（One-Dependent Estimator，简称 ODE）是半朴素分类中最常用的一种策略。所谓“独依赖估计”，也就是假设每个属性在分类类别之外最多仅依赖于一个其他属性。

与基于特征的条件独立性假设开展的朴素贝叶斯方法相比，其最大的区别就是半朴素贝叶斯算法放宽了条件独立假设的限制，考虑部分属性之间的相互依赖信息。但两者有共同特点：假设训练样本所有属性变量的值都已被观测到，即训练样本是完整的。

（3）高斯分布朴素贝叶斯计算方法

使用条件：所有特征向量都是连续型特征变量且符合高斯分布。

概率分布密度：

$$f(x)=\frac{1}{\sqrt{2\pi\delta}}\exp(-\frac{(x-\mu)^2}{2\delta^2})$$

（4）多项式朴素贝叶斯计算方法

使用条件：所有的特征变量都是离散型变量且符合多项式分布。

概率分布密度：

$$p(x=k)=C_n^k p^k(1-p)^{n-k}$$

（5）伯努利分布朴素贝叶斯计算方法

使用条件：所有的特征变量都是离散型变量且符合伯努利分布。

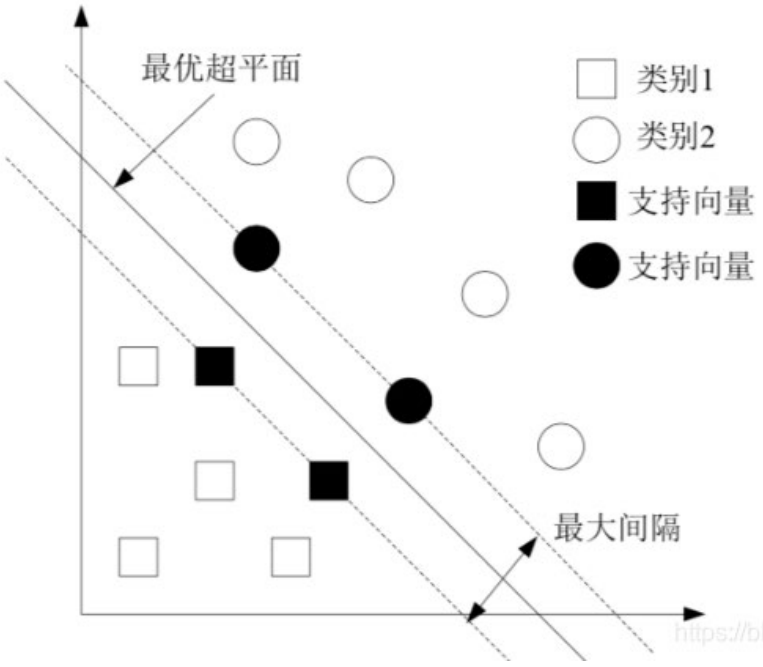
概率分布密度：

$$f(x)=\begin{cases}p, x=0\\ 1-p, x=1\end{cases}$$

(2) 支持向量机(SVM)

SVM 即支持向量机（Support Vector Machine)， 是有监督学习算法的一种，用于解决数据挖掘或模式 识别领域中数据分类问题。

二、基本原理：

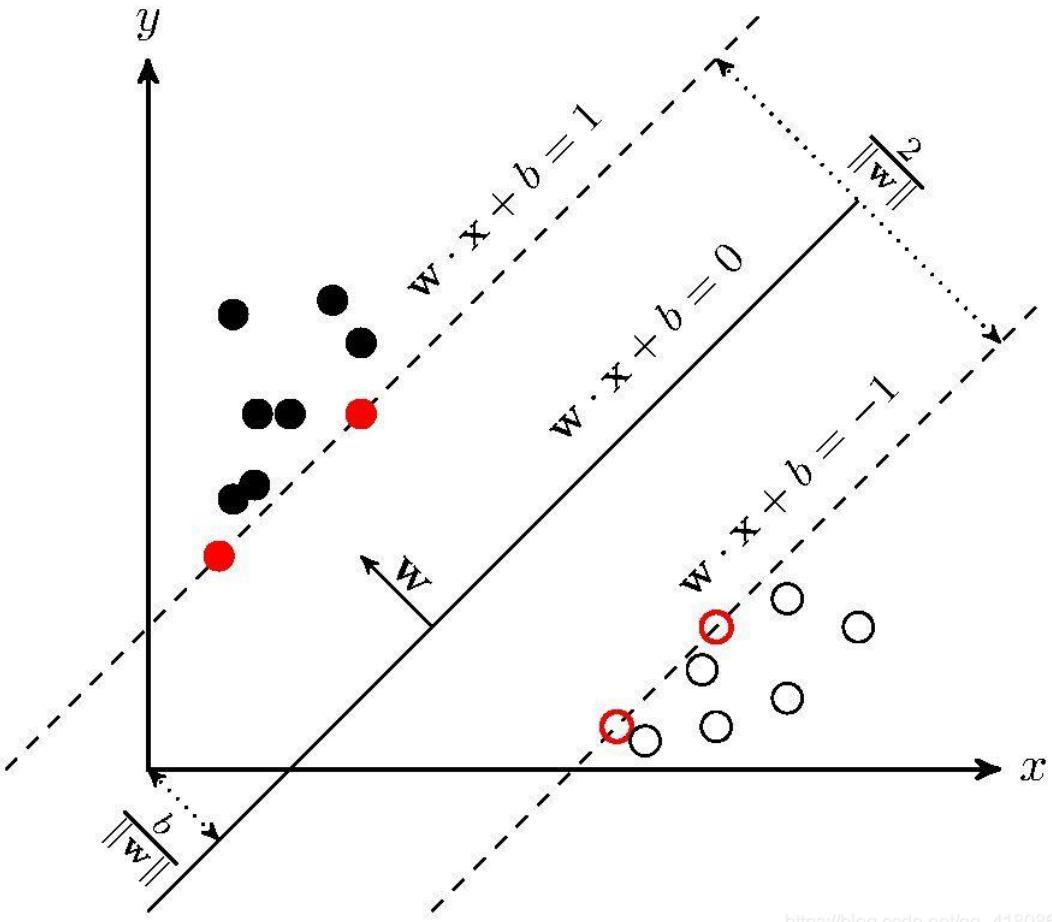


SVM 算法即寻找一个分类器使得超平面和最近的数 据点之间的分类边缘（超

平面和最近的数据点之间的 间隔被称为分类边缘）最大，对于 SVM 算法通常认为分类边缘越大，平面越优，通常定义具有“最大间 隔”的决策面就是 SVM 要寻找的最优解。并且最优 解对应两侧虚线要穿过的样本点，称为“支持向量”。其处理的基本思路为：把问题转化为一个凸二次规划 问题，可以用运筹学有关思想进行求解：①目标函数 在线性 SVM 算法中，目标函数显然就是那个"分类间 隔"，使分类间隔最大 ②约束条件 即决策面，通常需 要满足三个条件：1）确定决策面使其正确分类 2）决 策面在间隔区域的中轴线 3）如何确定支持向量 因此求解 SVM 问题即转化为求解凸二次规划的最优化问题。

支持向量机就是用来分割数据点那个分割面，他的位置是由支持向量确定的(如果支持 向量发生了变化，往往分割面的位置也会随之改变)，因此这个面就是一个支持向量确定的 分类器即支持向量机。

线性可分数据的二值分类机理：系统随机产生一个超平面并移动它，直到训练集中属于不同类别的样本点正好位于该超平面的两侧。显然，这种机理能够解决线性分类问题，但不能够保证产生的超平面是最优的。支持向量机建立的分类超平面能够在保证分类精度的同时， 使超平面两侧的空白区域最大化，从而实现对线性可分问题的最优分类。



SVM 的主要思想是：建立一个最优决策超平面，使得该平面两侧距平面最近的两类样 本之间的距离最大化，从而对分类问题提供良好的泛化力（推广能力）

“支持向量”：则是指训练集中的某些训练点，这些点最靠近分类决策面，是最难分类的数据点。

SVM：它是一种有监督（有导师）学习方法，即已知训练点的类别，求训练点和类别之间的对应关系，以便将训练集按照类别分开，或者是预测新的训练点所对应的类别。

(3) 多层感知器

多层感知机（MLP）是一种经典的神经网络模型，由多个神经元层组成。它的结构和功能使其成为深度学习中的重要组成部分。MLP 在各种任务中表现出色，如图像分类、文本分类、预测和回归等。

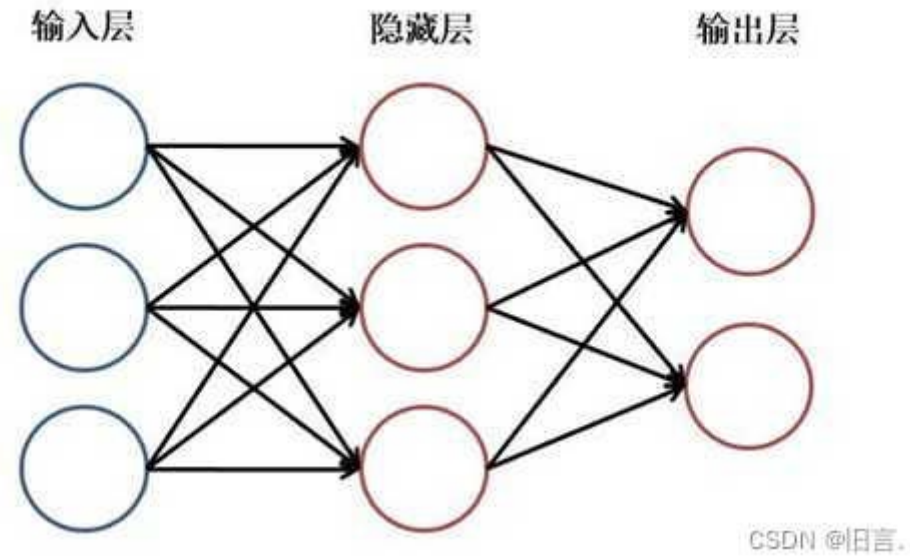
MLP 的原理

1.1 结构

MLP 由输入层、隐藏层和输出层组成。输入层接收输入数据，隐藏层通过学习特征表示，输出层产生最终的预测结果。隐藏层和输出层的每个神经元都具有激活函数，用于引入非线性映射。

1.2 激活函数

常用的激活函数包括 Sigmoid、ReLU、Tanh 等。激活函数的作用是在神经网络中引入非线性性质，使其能够学习复杂的非线性关系。



1.3 前向传播

MLP 的前向传播过程即从输入层到输出层的计算过程。它涉及到权重和偏置的计算、激活函数的应用等。通过一个简单的二分类任务为例来演示 MLP 的前向传播过程。

1.4 反向传播算法

反向传播算法是用于训练 MLP 模型的关键步骤。通过计算梯度来调整权重和偏置，以最小化预测结果与真实结果之间的误差。在本文中，将使用反向传播算法来训练 MLP 模型并进行分类任务。

(4) 主成分分析法

主成分分析法（Principal Component Analysis，PCA）是一种常用的数据分析技术，用于降维和特征提取。它的基本原理是通过线性变换，将高维数据转换为低维空间，同时保留数据的最大方差。主成分分析也被广泛应用于数据可视化、

数据压缩、特征选择和模式识别等领域。

1. 数据标准化

在进行主成分分析之前，需要对原始数据进行标准化处理。这是因为各个特征可能具有不同的尺度和变化范围，而 PCA 是基于数据的协方差矩阵计算的，所以标准化可以消除单位和尺度的差异。

2. 协方差矩阵计算

通过对数据进行标准化，得到一个标准化后的数据矩阵。然后计算该数据矩阵的协方差矩阵。协方差矩阵描述了数据中不同特征之间的相关性。

3. 特征值分解

对协方差矩阵进行特征值分解，得到特征值和对应的特征向量。特征值代表了每个特征方向上的方差，而特征向量则表示了相应特征方向的权重。

4. 特征值排序和选择

将特征值按照大小进行排序，选择最大的 k 个特征值对应的特征向量作为主成分。这些主成分对应了最大的方差，即数据中最重要的特征。

5. 数据投影

将原始数据投影到选定的主成分上，得到一个新的低维数据矩阵。投影过程可以看作是将原始数据在主成分上的投影的线性组合。

通过这些步骤，主成分分析法实现了对高维数据的降维和特征提取。主成分分析还具有对数据进行可视化的能力，可以通过绘制主成分的散点图或者数据的散点图来展示数据的分布和相关性。

主成分分析法的应用非常广泛。它可以用于数据压缩，在保持数据相对完整性的情况下减少数据的存储空间。主成分分析还可以用于特征选择，帮助识别最具有代表性的特征。此外，在模式识别和机器学习任务中，主成分分析可作为预处理步骤，提取最具有区分性的特征。

主成分分析法是一种常用的数据分析技术，通过线性变换将高维数据转换为低维空间，保留最大的方差。通过深入理解主成分分析法的基本原理和步骤，我们可以更好地应用它进行数据降维和特征提取，从而更好地理解和分析数据。

曲线拟合 APP，是基于最小二乘法原理，将一组数据通过选定的数据拟合算法拟合成一组曲线，选择适当的曲线类型来拟合观测数据，并用拟合的曲线方程分析两变量间的关系。

三、实验结果与分析

数据集介绍： 一个血液各项指标与其对应的疾病组成的数据集(train:2351,test:486)

各项指标包括(共 24 项)：

属性名称	中文翻译
Glucose	血糖
Cholesterol	胆固醇
Hemocyte	血细胞

Platelet	血小板
Wite Blood Cells	白细胞
Hematocrit	红细胞
Mean Corpular Volume	血红蛋白
Mean Corpular Hemo	均体体积
Mean Corpular Hemoglobin Concentration	均体血红蛋白
Insulin	平均体血红蛋白浓度
BMI	胰岛素
Systolic Blood Pressure	身体质量指数
Diastolic Blood Pressure	收缩压
Triglycerides	舒张压
HbA1c	三脂
LDL Cholesterol	糖化血红蛋白
HDL Cholesterol	高密度脂蛋白
ALT	低密度脂蛋白
AST	谷丙转氨酶
Heart Rate	心率
Creatinine	肌酐
Troponin	心肌肌钙蛋白
C-reactive Protein	C 反应蛋白

图 1 数据集各项指标及分布

疾病类型包括（共 5 项）：

Healthy: 正常

Diabetes: 糖尿病

Thalasse Disease: 地中海贫血

Anemia: 贫血

Thromboc: 甲状腺疾

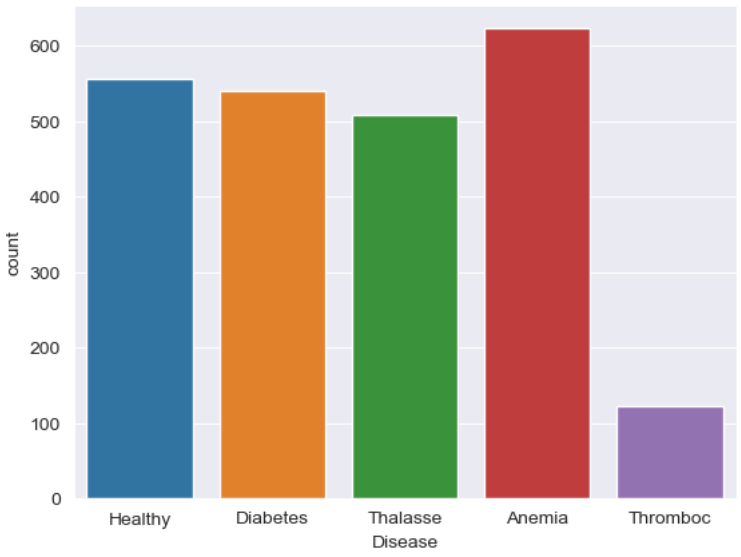


图 2 疾病种类及数据量

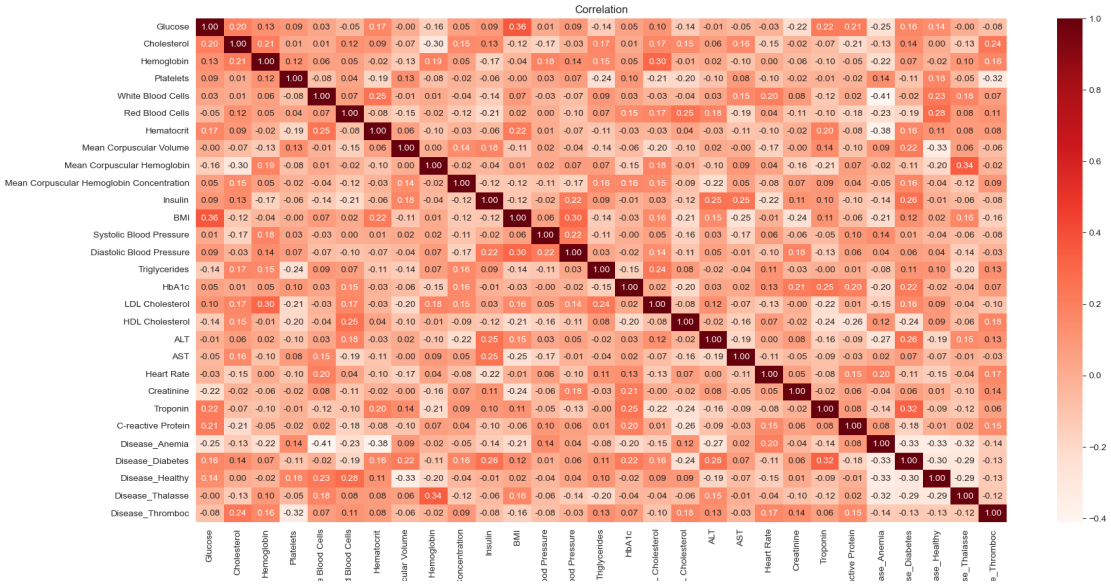


图 3 各项指标相关矩阵

1. 支持向量机

在 SVC 中分别使用了不同核函数，其效果也不尽相同

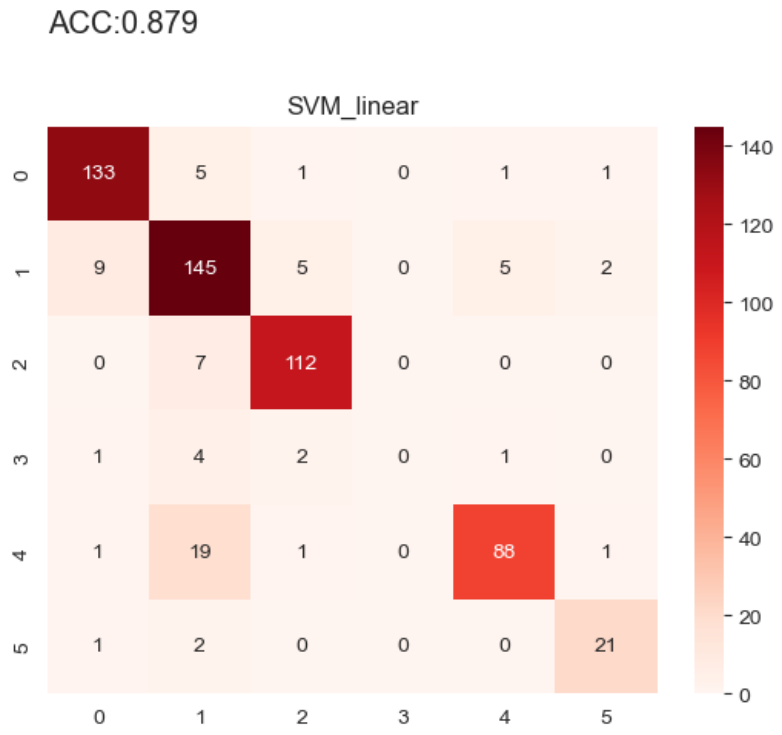


图 4：SVM 线性核函数混淆矩阵 ACC:87.9%

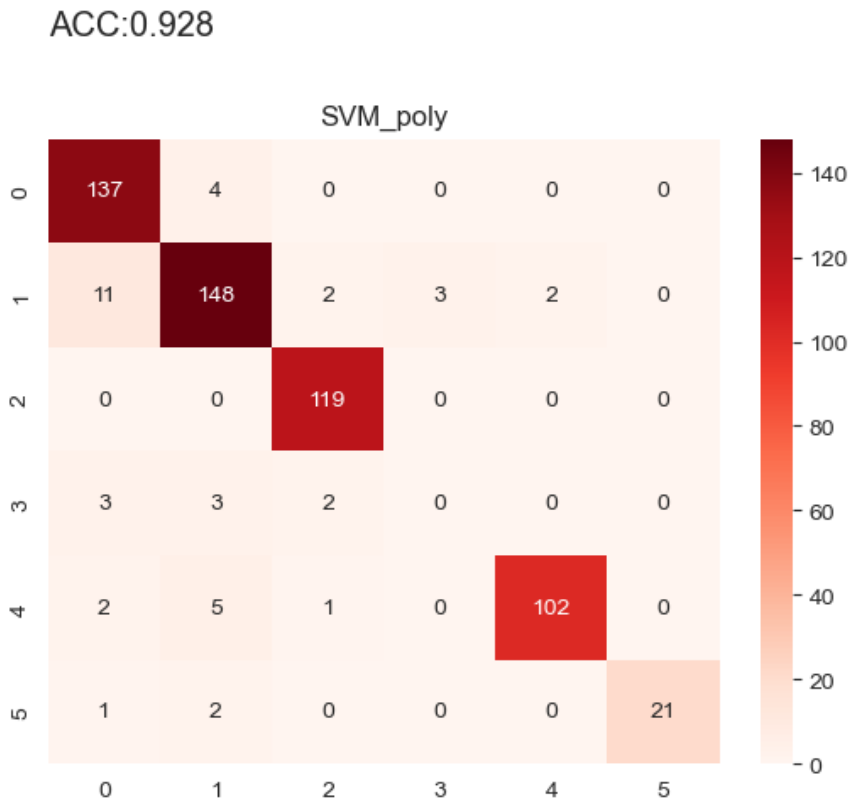


图 5：SVM 多项式核函数混淆矩阵 ACC:92.8%

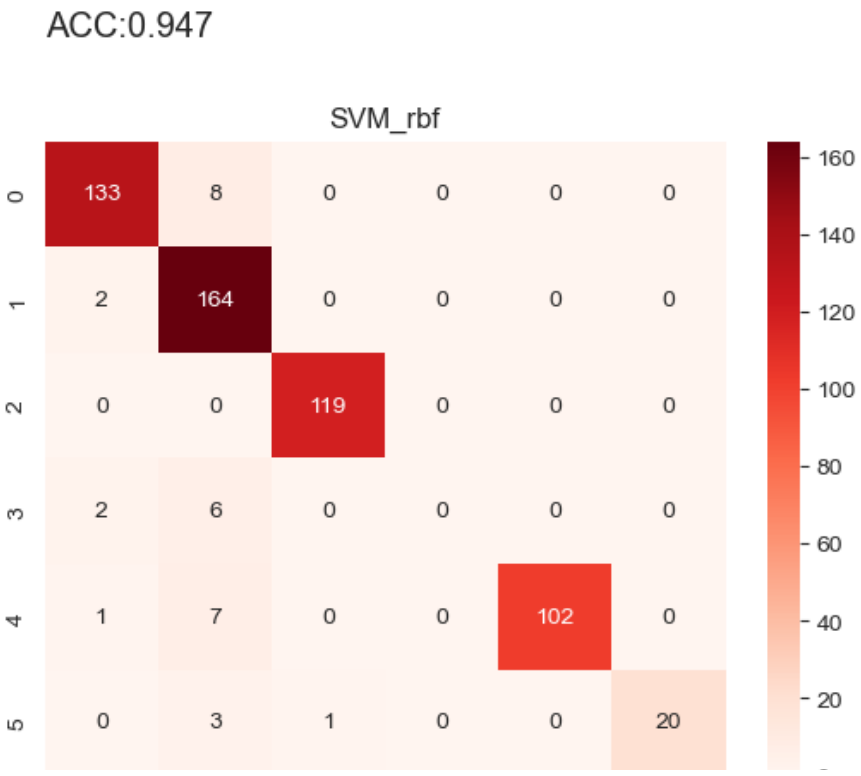


图 6：SVM 径向基核函数混淆矩阵 ACC:94.7%

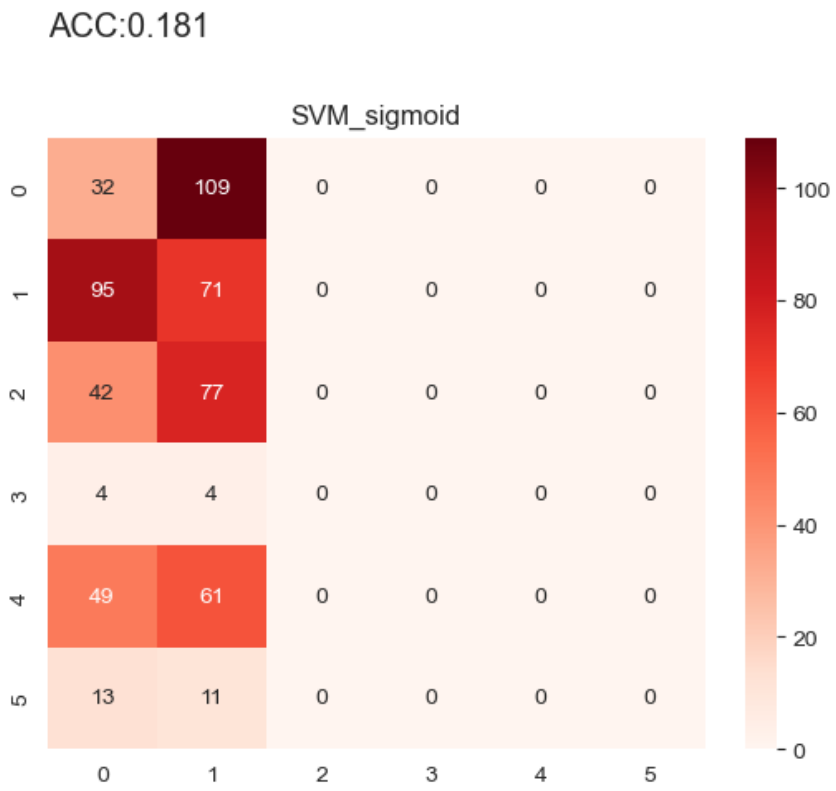


图 7：SVM sigmoid 核函数混淆矩阵 ACC:18.1%

2. 贝叶斯

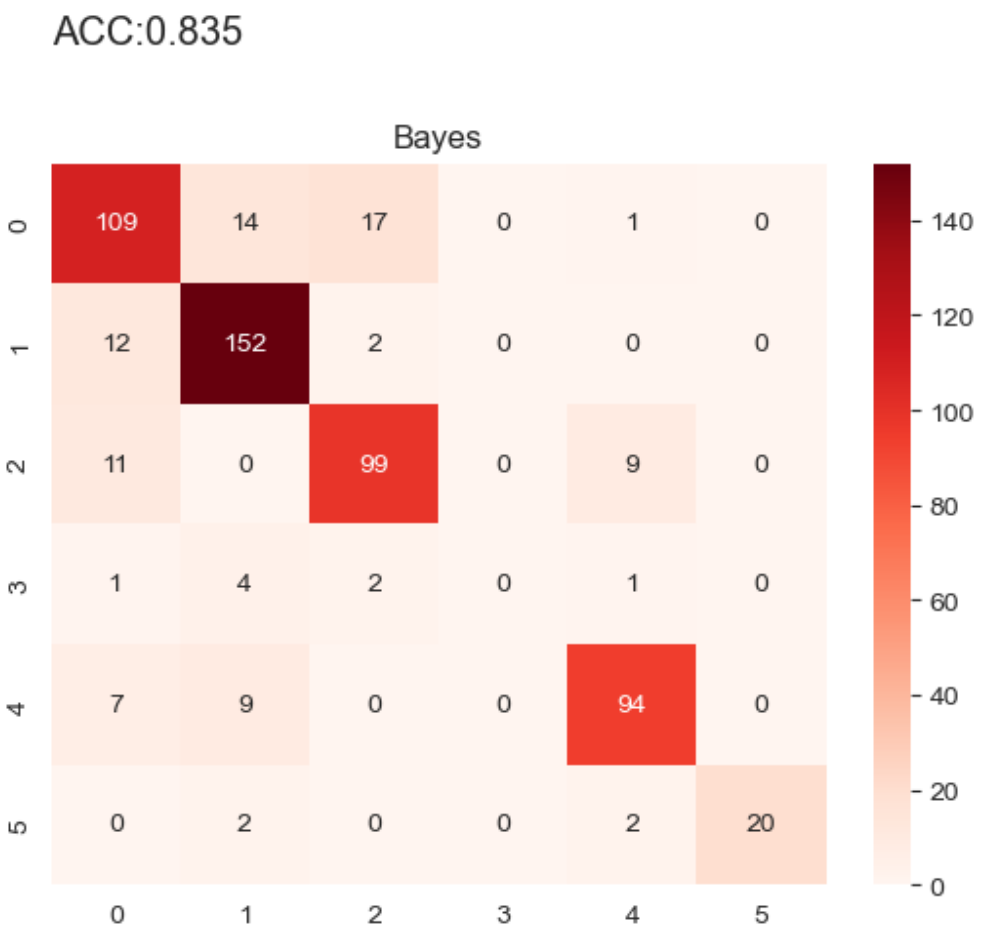


图 7：朴素贝叶斯分类器混淆矩阵 ACC:83.5%

3. 多层感知器(sklearn)

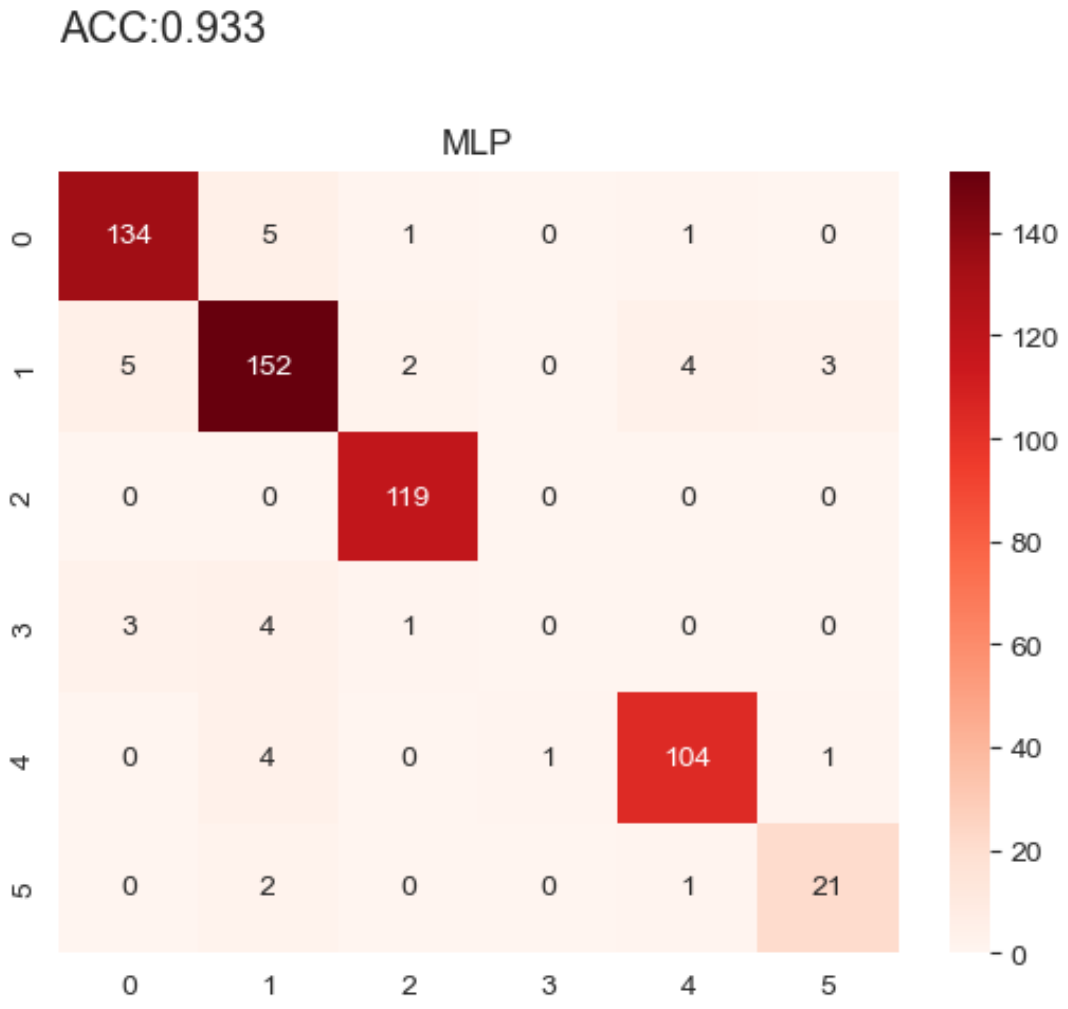


图 7： 多层感知器混淆矩阵 ACC:93.3%

4. 主成成分分析 + SVM

ACC:0.947

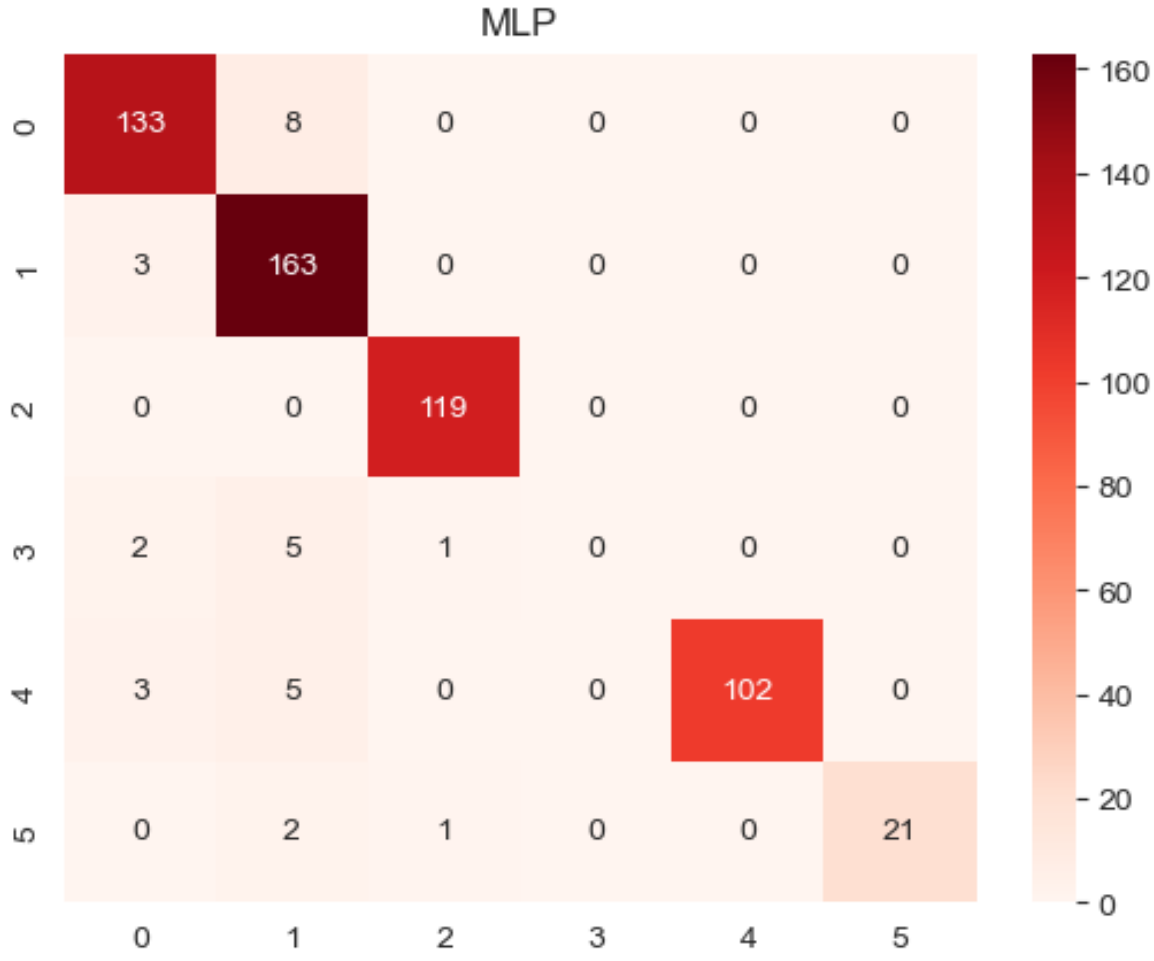


图 7：SVM(rbf)+PCA(0.9)混淆矩阵 ACC:94.7%

5. 主成成分分析 + 贝叶斯

ACC:0.808

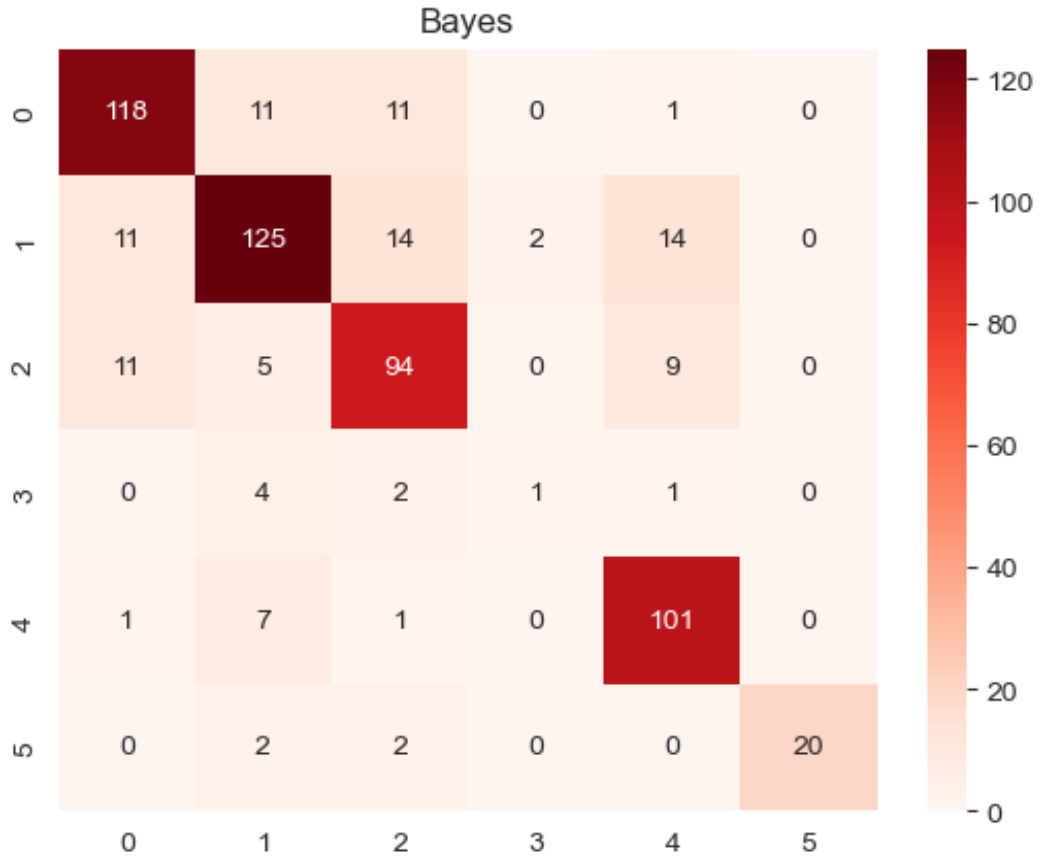


图 7：PCA(0.9) + 贝叶斯混淆矩阵 ACC:80.8%

6. 多层感知器(Pytorch)

层定义:

Num	层大小
1	Linear(24, 256)
2	Linear(256, 256)
3	Linear(256, 100)
4	Linear(100, 6)

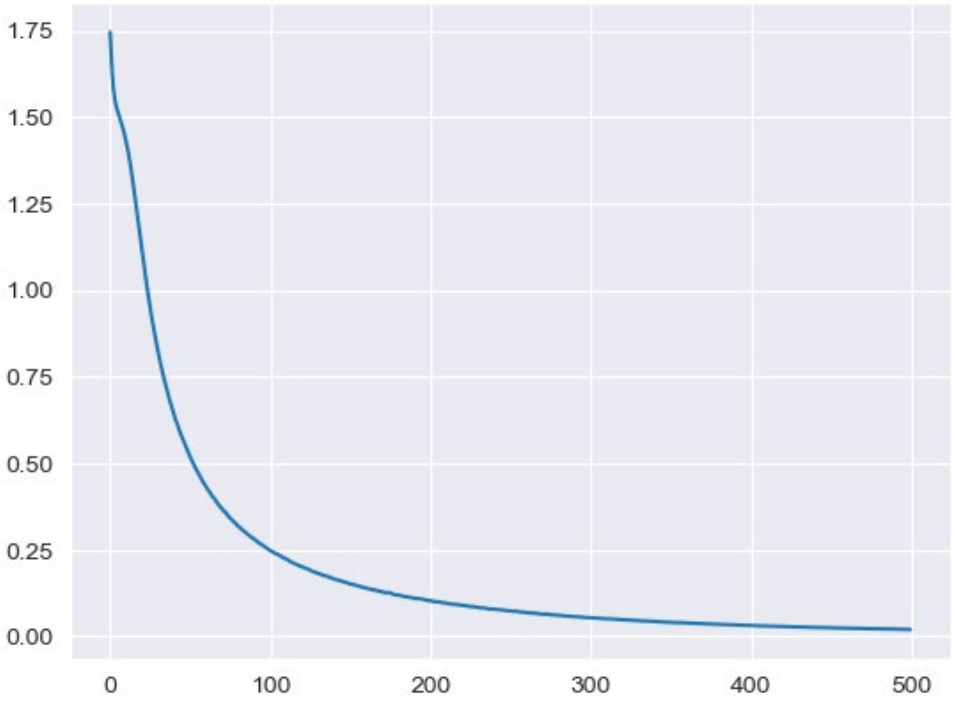


图 7: 损失函数下降曲线

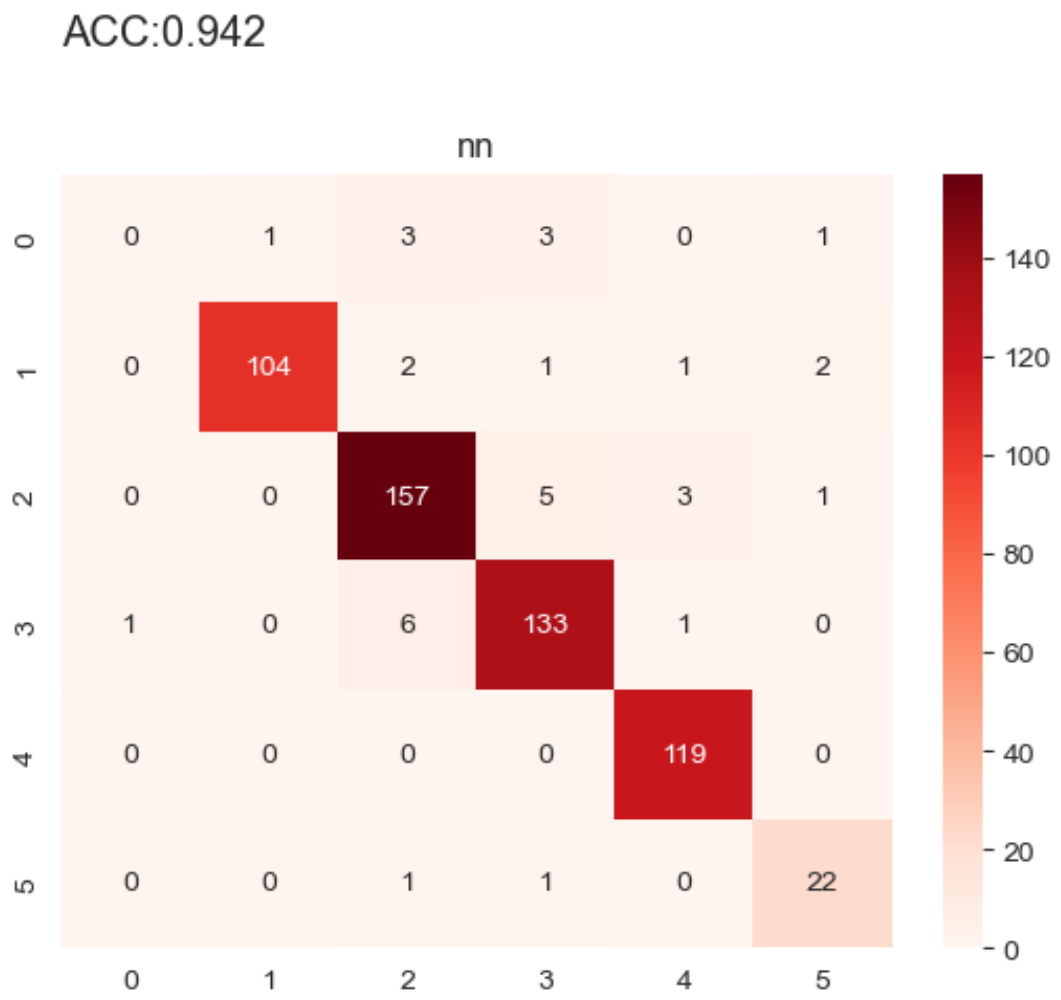


图 7： 自己构建网络混淆矩阵 ACC:0.942

Epoch:500 Lr:0.001 momentum:0.9

四、结论

模型名称	准确率
SVM(线性核)	0.879
SVM(多项式核)	0.928
SVM(径向基核)	0.947
SVM(Sigmoid 核)	0.181
贝叶斯	0.835
多层感知器	0.933
PCA + SVM	0.947
PCA + 贝叶斯	0.808
自定义多层感知器	0.942

在 SVM 中径向基核的表现最好，准确率位 0.947，在后来的实验中，我加入了 PCA 主成成分分析通过修改保留特征的参数，分类器可以表现得更好，但是参数过低或者过高可能会造成效果反噬

贝叶斯分类器没有如此表现，我通过调整参数，并没有使得准确率提升，反而造成了下降

但是可能在某些方面得到了提升，比如说

不同分类器中准确率的情况：SVM > MLP > 贝叶斯

其中最贴近 SVM(rbf)[ACC:0.947]的最高准确率的算法是自定义的 MLP[ACC:0.942]

SVM 确实是一种强大的算法，因为只取决于支持向量，所以计算量很

少，对与非线性问题，还可以用核函数来解决，是深度学习没有兴起之前最🐂的算法了

但现在流行的 MLP 我的看法，他很暴力很无脑适用于解决各种问题

总的来说，深度学习与机器学习算法都能很好的解决该数据集的分类，并且有着不错的效果.

五、核心程序（含注释）

具体代码放在我的 github 上了：
https://github.com/wushenhaoyu/NWPU-Pattern-recognition.git
大作业在 jupyter notebook 上编写

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.decomposition import PCA
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier #导入包

train = pd.read_csv('Blood_samples_dataset_balanced_2(f).csv')  #导入训练集
test = pd.read_csv('blood_samples_dataset_test.csv')  #导入测试集

print(train.shape) #打印训练集矩阵
print(test.shape) #打印测试集矩阵

train.head() #打印训练集前几个样本情况

train.hist(figsize=(20, 10)) #绘制训练集样本各属性分布情况
plt.show()

sns.countplot(data= train, x= 'Disease') #绘制训练集样本分类情况
plt.show()
```

```
encoded = pd.get_dummies(train)
cor = encoded.corr()
plt.figure(figsize=(20, 10))
sns.heatmap(cor, annot=True, cmap = 'Reds', fmt=".2f")
plt.title('Correlation')
plt.show()  #绘制总体样本相关矩阵
df = pd.concat([train,test],ignore_index=True)
X = df.drop(columns=['Disease'])
y = df['Disease']
#%%
X_train,  X_test,  y_train,  y_test  =  train_test_split(X,  y,
test_size=0.2, random_state=42) #构建测试#和训练数据集并打乱

def calculate_acc(confusion_matrix):  #定义准确率计算函数
    correct =  np.trace(confusion_matrix)
    total = np.sum(confusion_matrix)
    return correct / total
#%%
def svm(kernel):  #定义使用所有 SVM 核函数
    SVM = SVC(kernel=kernel)
    SVM.fit(X_train, y_train)
    return SVM.predict(X_test)

kernels = ['linear', 'poly', 'rbf', 'sigmoid']
#%%
for index, i in enumerate(kernels): #依次进行 SVM 并绘制相应混淆矩阵
    cm = confusion_matrix(y_test, svm(i))
    sns.heatmap(cm, annot=True, cmap = 'Reds', fmt="d")
    plt.title('SVM_'+ i)
    plt.annotate("ACC:%.3f"%(calculate_acc(cm)),       (0,       -1),
fontsize="x-large", annotation_clip=False)
    plt.show()


bayes = GaussianNB() #初始化贝叶斯并计算绘制混淆矩阵
#%%
bayes.fit(X_train, y_train)
cm = confusion_matrix(y_test, bayes.predict(X_test))
sns.heatmap(cm, annot=True, cmap = 'Reds', fmt="d")
plt.title('Bayes')
plt.annotate("ACC:%.3f"%(calculate_acc(cm)),  (0,  -1),  fontsize="x-
large", annotation_clip=False)
plt.show()
```

```
#初始化多层感知器并计算绘制混淆矩阵
MLP = MLPClassifier(hidden_layer_sizes=[20,100],max_iter=1000,random_state=62)
MLP.fit(X_train, y_train)
cm = confusion_matrix(y_test, MLP.predict(X_test))
sns.heatmap(cm, annot=True, cmap = 'Reds', fmt="d")
plt.annotate("ACC:%.3f"%(calculate_acc(cm)), (0, -1), fontsize="x-large", annotation_clip=False)
plt.title('MLP')
plt.show()

#先对数据进行 PCA 降维并初始化 SVM（径向基核函数）并计算绘制混淆矩阵
pca = PCA(n_components=0.9)
svm = SVC(kernel='rbf')
pipe = make_pipeline(pca, svm)
pipe.fit(X_train, y_train)
cm = confusion_matrix(y_test, pipe.predict(X_test))
sns.heatmap(cm, annot=True, cmap = 'Reds', fmt="d")
plt.annotate("ACC:%.3f"%(calculate_acc(cm)), (0, -1), fontsize="x-large", annotation_clip=False)
plt.title('MLP')
plt.show()

#先对数据进行 PCA 降维并初始化 MLP 并计算绘制混淆矩阵
bayes = GaussianNB()
pca = PCA(n_components=0.9)
pipe = make_pipeline(pca, bayes)
pipe.fit(X_train, y_train)
cm = confusion_matrix(y_test, pipe.predict(X_test))
sns.heatmap(cm, annot=True, cmap = 'Reds', fmt="d")
plt.annotate("ACC:%.3f"%(calculate_acc(cm)), (0, -1), fontsize="x-large", annotation_clip=False)
plt.title('Bayes')
plt.show()

#自己构建 pytorch 多层感知器并梯度下降训练分类器
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
import torch.optim as optim
#%%
```

```
class Mydatasets(Dataset): #构建数据类
    def __init__(self, X, y):
        label_to_index = {label: index for index, label in enumerate(set(y))}
        self.data = X
        self.label = [label_to_index[label] for label in y]
    def __getitem__(self, index):
        if index >= len(self.data):
            raise IndexError(f"Index {index} is out of bounds for dataset of size {len(self.data)}")
        return torch.tensor(self.data.iloc[index].values, dtype=torch.float32), torch.tensor(self.label[index], dtype=torch.int64)
    def __len__(self):
        return len(self.data)
#%%
class classifier(nn.Module): #构建分类器
    def __init__(self):
        super(classifier, self).__init__()
        self.fc1 = nn.Linear(24,256)
        self.fc2 = nn.Linear(256, 256)
        self.fc2 = nn.Linear(256, 100)
        self.fc3 = nn.Linear(100, 6)
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
#%%
#使用数据类放入数据
trainloader = DataLoader(Mydatasets(X_train, y_train), batch_size=32, shuffle=True)
testloader = DataLoader(Mydatasets(X_test, y_test), batch_size=32, shuffle=False)

#判断有没有 cuda 有就用 GPU 加速没有用 CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#%%
# 初始化分类器并定义损失函数 优化器
net = classifier().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
list=[]
#minibatch 训练
```

```
for epoch in range(500):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {running_loss/len(trainloader)}")
    list.append(running_loss/len(trainloader))
plt.plot(list) #绘制损失函数梯度下降曲线

def check():#定义检查函数进行无梯度计算预测并绘制混淆矩阵
    total = 0
    correct = 0
    all_preds = []
    all_labels = []

    net.eval()
    with torch.no_grad():
        for data in testloader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
    cm = confusion_matrix(all_labels, all_preds)

    sns.heatmap(cm, annot=True, cmap = 'Reds', fmt="d")
    plt.title('nn')
    plt.annotate("ACC:%.3f"%(calculate_acc(cm)),        (0,        -1),
fontsize="x-large", annotation_clip=False)
    plt.show()

#%%
check()#展示图像
```