AdaBoost-KNN Project Report

Author: Maythics
ID: unknown

Repository

本次作业的 notebook 代码以及报告等相关内容均已经上传至我的仓库，如有疑问请访问 https://github.com/Maythics/AdaKNN-report

INTRODUCTION

Method

概述方法：Adaboost 本来用于二分裂问题，但是 KNN 则是针对多分类（KNN 二分类也没什么意思），因此需要知道如何让 Adaboost 支持多分类，下面是（详见 reference）简单陈述：

对于多维，将样本的标签也设置为多维的，比如：$y = \begin{cases} 1 & y \in k \text{ family} \\ -\frac{1}{k-1} & \text{else} \end{cases}$

修改传统的 Adaboost 中的指数损失函数为：

$$L(y, f(x)) = \sum_i \exp\left(-\frac{1}{k}\vec{y}_i \cdot \vec{f}_{m(x_i)}\right)$$

把当前的分类器拆分成之前的加上最近一步的：

$$f_{m(x)} = f_{m-1}(x) + \alpha_m g_{m(x)}$$

然后代入，看看损失函数在正确与否的情况下分别是多少，得到一个能用事情函数表达出来的式子，再对α求偏导，思路和李航书中 Adaboost 相同……

最终，得到的算法如下：

1. 初始化权重w_i
2. 在权重w_i下训练分类器
3. 计算错误率error $= \sum_i w_i \frac{I(g_m \neq \vec{y}_i)}{\sum_i w_i}$
4. 计算子分类器权重$\alpha_m = \ln\left(\frac{1-\text{error}}{\text{error}}\right) + \ln(N-1)$，$N$是类别数
5. 更新样本点权重$w_i = w_i \exp(\alpha_m I(g_m \neq \vec{y}_i))$
6. 归一化权重，然后循环往复

下面还要解决一个问题，就是如何将 KNN 用于优化，因为 KNN 基于的是固定的 k 个最近邻，好像没什么参数可以调的，于是这里使用的是 weigthed KNN，这样就有权重了

具体含义

例子：比如 $k = 3$，则先找三个最邻近的点，它们权重分别为 A(0.3)，A(0.6)，B(0.7)，0.3+0.6=0.9>0.7 因此，以最后选择归到 A 类中

可以看出，其实这就是一个投票池系统，每来一个新的点，就先找 k 个最近，然后再发动这 k 个代表带权投票，投票值最大的类胜出；再来一个点，再发动一次找最近，最近的几个再带权投票……

在本 Project 中，训练时采用的是随机滚动洗牌的方法，输入某些带有标签的数据后，这样训练：

1. 第一轮，先人为地从数据中分出一个测试集，剩余的当模型点集
2. 根据模型点集中的点，尝试用 weighted KNN 分类测试集
3. 根据在测试集上的表现，更新测试集的权重（按照上面的 Adaboost）
4. 归一化权重，我这里是将训练集+模型点集一起归一化
5. 将模型集与训练集合并，然后洗牌，重新分出测试集和模型集，重复

在某一轮分错的点，其权重会增大，之后的几轮中，它被洗牌到模型集之后，就会在投票时获得更大的票数。也就是，“之前的分错会导致之后投票时话语权更大”，这样就实现了 weighted KNN 的变权重，之后会更加照顾分错点的意见

Reference

我参考了两篇文献以及 CSDN 论坛

一篇是改进的 AdaBoost 集成学习方法研究[D].暨南大学,2021.DOI:10.27167/d.cnki.gjinu.2020.001350.

一篇是 Zhu, H. Zou, S. Rosset, T. Hastie, “Multi-class AdaBoost”, 2009.

CSDN 内容：https://blog.csdn.net/weixin_43298886/article/details/110927084

CODES AND RESULTS

experiment environment

我使用 Anaconda 中的环境，使用 jupyternotebook 进行编辑，使用了 **torch** 库，其版本为 2.2.1

```
[23]: print("PyTorch 版本:", torch.__version__)
      PyTorch 版本: 2.2.1
```

Figure 1: 检查版本

Codes

代码如下：

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class KNN(nn.Module):
    def __init__(self, k):
        super(KNN, self).__init__()
        self.k = k

    def forward(self, x_train, y_train, x_test,w_train):
        dists = torch.cdist(x_test, x_train)  # 计算测试样本与训练样本之间的距离，输出是：第一行是 test 中的第一个和 train 中的每一个分别的欧氏距离，第二行是 test 中第二个和 train 中每一个的欧氏距离
        _, indices = torch.topk(dists, self.k, largest=False)  # 选择最近的 K 个样本的索引，存在 indices 中

        knn_labels = y_train[indices]  # 获取最近的 K 个样本的标签
        # print(y_train[indices])
```

```
        vote_pool=torch.zeros(x_test.size()[0],x_train.size()[0])  # 新建一个投票池，列数本来应该是类的个数，但因为未知类的个数，不妨就取 train 几个样本，就有几个列，因为样本数一定大于类的个数，因此，类别数不能乱写，比如一共就 4 个样本，不能有样本是第 9 类的
        #print("vote init",vote_pool)

        #print("hey",indices.size()[0])
        for j in range(0, indices.size()[0]):
            # print("this is j",j)

            for i in indices[j]:  # 对于每一个 x_test 都有各自的投票池，这是给第 j 个池子投票
                #print("this",i)   #i 的含义就是哪几个是 k 个最近点
                vote_pool[j][y_train[i]] += w_train[i] # y_train[i]表示 i 所在类的那个指标，往这个类的投票箱子里投权重那么多票

        #print("vote number",vote_pool)
        # 下面开始每一行各归某一行统票
        max_indices = torch.argmax(vote_pool, dim=1) #torch.argmax 对于 dim=1 这个维度考察，找到票数最多的，存入 1 行多列的 max_indices 中

        #print("max_indices",max_indices)

        # 这个 max_indices 代表胜出的类，第一个分量就是 x_test 中的第一个点应该归到的类，第二个分量就是 x_test 中第二个点应该归到的类
        return max_indices


def adjust_weight(pred_labels,y_test,w_train,w_test,N):   # 传入参数 N，即分类的总个数，相当于上图中的 N
    # 方法是，根据上面得到的 pred_labels 和我的 y_test 一个分量一个分量对比过去，如果第 i 个分量不同，就代表第 i 个数据是归类错误的
    error_index=torch.nonzero(pred_labels-y_test)
   # print("error_index",error_index)
    # 先计算分类错误率，error（就是上图中的 r_error）
    numerator=0   # 分子先设置为 0
    for i in error_index:
      #print("index=",i.item())
      #print("w_test[i.item()]",w_test[i.item()])
      numerator += w_test[i.item()]
      # print("numerator is",numerator)

    error = numerator/sum(w_test).item()  # 分母是 test 集中的权重之和，分子是 tets 集里分错的权重之和
    print("error",error)

    # 下面计算 alpha
    if error >0 and error < 1 :
      alpha = math.log((1-error)/error)+math.log(N-1)   # 这里的几段是为了防止分母为 0 的报错，error 得在 0,1 中间
    #print("alpha",alpha)
    elif error==1:
        alpha=0
    else:           # 此时 error=0，证明训练很好了，给它安排较大的权重，比如 10
        alpha = 1000
    # 下面更新测试的样本权重，仅仅对于分错的遍历就行了，因为分对的那些权重是不变的（见上面的公式）
    for i in error_index:
```

```
      w_test[i.item()]=w_test[i.item()]*math.exp(alpha)
    #print("new wieght",w_test)

    # 下面归一化权重，此时是将所有的样本一起归一化，要包括 train 和 test 两者
    totalsum=sum(w_train)+sum(w_test)
    w_train=w_train/totalsum
    w_test=w_test/totalsum
    #print("new wieght",w_train,w_test)
    return w_train, w_test, alpha

# 先打乱
def generate_test_and_train(x_data,y_data,w_data):
    permuted_indices = torch.randperm(x_data.size(0)) # 这里取其他的几个也是可以的，反正列数都是一样的
    #print(permuted_indices)

    # 使用这个排列对张量进行重新排列
    x_data = x_data[permuted_indices]
    y_data = y_data[permuted_indices]
    w_data = w_data[permuted_indices]

    # 打乱后把前 3 个分出去，代表 test，剩下的代表 train

    x_train = x_data[4:]
    y_train = y_data[4:]
    w_train = w_data[4:]

    x_test = x_data[:4]
    y_test = y_data[:4]
    w_test = w_data[:4]

    #print("随机打乱第 0 维度后的张量: \n", x_data,y_data,w_data)
    #print(x_test,x_train)
    #print(y_test,y_train)
    return x_train,x_test,y_train,y_test,w_train,w_test # 输出一共 6 个变量

################################### 主函数 ###################################

class ModelStructure:
    def __init__(self, tensor1, tensor2, tensor3,tensor4):
        self.tensor1 = tensor1
        self.tensor2 = tensor2
        self.tensor3 = tensor3
        self.tensor4 = tensor4

classifier_array = []

def Classify(classifier_array,x_data,y_data,knn):
    # 构造一个大投票池
    big_vote_pool = torch.zeros(x_data.size()[0],3) #x_data().size()[0]行 3 列，3 其实是分的类别数，每
一行代表考察的样本，那一行就是他的投票池

    for subclassifier in classifier_array:
      pred_labels = knn(subclassifier.tensor1, subclassifier.tensor2, x_data ,
subclassifier.tensor3)
```

```
        # 这里，带预测的数据就是 x_data 原本的样本集，而其他三个位置分别是 classifier_array 中的某一个结构体中的几个张量
        #print("pred_labels",pred_labels,"alpha",subclassifier.tensor4)
        for row in range(0,x_data.size()[0]):  # 对于每一个样本遍历，此时是某一个固定的子分类器上台投票
            big_vote_pool[row][pred_labels[row]] += subclassifier.tensor4 # 投出大小为 alpha 的一票

    #print(big_vote_pool)
    final_decision = torch.argmax(big_vote_pool, dim=1)
    #print(final_decision)
    predict_error_index=torch.nonzero(final_decision-y_data)
    print(predict_error_index.size(0))
    print("error rate",predict_error_index.size(0)/final_decision.size(0))



def main():
    # 这是原始输入数据，请在此自拟
    x_data = torch.tensor([[1.0, -2.0], [-2.0, 3.0], [3.0, 4.0], [1.2, -0.5],[-3.2,4.5],[0.02, 3.5], [1.0, 2.0],[4,12],[10,-2],[3,14],[2.7,6],[5,1],[-4,2],[6,-2],[-0.5,2],[2,8],
                           [3,0],[2,1],[-2,2],[-1.5,5],[2,4.7],[3.8,10],[0.5,3],[0.6,1.8],[2,3],[0.2,1.5],[-0.5,1],[-0.8,0.8],[-1,2.1]])
    y_data = torch.tensor([0,2,1,0,2,  2,1,1,0,1,  1,0,2,0,2,1, 0,0,2,2,1, 1,2,1,1,1,1,1,2])
    w_data = torch.tensor([1,1,1,1,1,  1,1,1,1,1,  1,1,1,1,1,1, 1,1,1,1,1, 1,1,1,1,1,1,1,1]) # 初始权重，可以自己设置

    knn = KNN(3) #knn 中 k 取几

    for cnt in range(0,5):

        x_train,x_test,y_train,y_test,w_train,w_test = generate_test_and_train(x_data,y_data,w_data)

        pred_labels = knn(x_train, y_train, x_test,w_train)

        w_train, w_test,alpha = adjust_weight(pred_labels,y_test,w_train,w_test,3) # 一共 3 类进行测试
        #print("weight_train",w_train,"weight_test",w_test)

        # 把这个子分类器存起来，存在一个 classifier[]内部，要的就是 x_train,y_train,w_train 和 alpha，这几个量表征了这个子分类器
        classifier_array.append(ModelStructure(x_train,y_train,w_train,alpha))

        # 把数据集和训练集重新拼起来，以便下一轮打乱

        x_data = torch.cat((x_train, x_test), dim=0)
        y_data = torch.cat((y_train, y_test), dim=0)
        w_data = torch.cat((w_train, w_test), dim=0)
        # print(x_data,y_data,w_data )

    print("\n\n Here is my classifier\n")
    for item in classifier_array:
        print("x_train:", item.tensor1)
        print("y_train:", item.tensor2)
        print("w_train:", item.tensor3)
```

```
        print("alpha:", item.tensor4)
    # 预测，试试看效果
    # 这是在原始数据集上的预测表现
    print("----------in the original data set, the prediction result is---------")
    Classify(classifier_array,x_data,y_data,knn)

    print("----------now lets test it with new input dots, the prediction result is---------")
    # 接下来这两个是全新的样本点，与之前是毫无关系的，纯粹是看看能否预测
    testdata_x = torch.tensor([[1.0, -1.0], [-2.0, 1.2], [3.0, 10], [1.2, 0],[-1,2],  [1.0,
-1.5], [-1.7, 0.5], [3, 3], [1, 0.1],[-1,2.4]])
    testdata_y = torch.tensor([0,2,1,0,2,0,1,1,0,2])
    Classify(classifier_array,testdata_x ,testdata_y ,knn)
main()
```

Explanation

代码解释：主要的函数是这几个：

1. KNN 与 knn，这个类是用来生成某一个 knn 的，比如：`knn = KNN(3)`就能创建一个找出 3 个最近点的函数 knn，而 knn 函数的输入是模型点集和其标签和其权重（x_train, y_train,w_train），以及无标签的预测点集（x_test），返回的是一个张量 pred_labels，存有个根据这个 weighted knn 得出的点集 x_test 预测出的标签结果
2. adjust_weight，该函数是用来更新权重，算出 alpha 之类的数据的。输入是之前预测点集的标签 pred_labels，以及预测点集实际的标签 y_test，模型点集目前的权重 w_train，测试点集目前的权重 w_test，还有本次分类的类别数 N；输出是更新后的模型点集的权重 w_train，更新后的测试点集权重 w_test，以及本次更新对应的 alpha
3. generate_test_and_train 该函数用于洗牌，重新划分谁是模型点集，谁是测试点集，输入就是整合后的点集和其标签和其目前的权重（x_data 以及 y_data,w_data)
4. main 主函数，首先是输入数据 x_data，y_data，w_data 的给出，然后是产生一个 knn 函数。下面开始循环：每一次都先调用 generate_test_and_train 函数，根据 x_data，y_data，w_data 得到本轮的模型集和测试集，然后把它们输入 knn 中，得到测试集的预测结果，接着要根据预测的好坏调整权重，调用 adjust_weight 函数，得到更新后的权重。最后，把新权重、点集、标签全部再合并起来（torch.cat）得到新的 x_data，y_data，w_data（和之前的区别就是权重已更新，而且顺序不同），开始下一轮

Data set

我采用了如下图所示的训练样本点(x_data 与 y_data)，其中不同颜色代表不同的类别，绿色代表 0 类，红色代表 1 类，蓝色代表 2 类

```
x_data = torch.tensor([[1.0, -2.0], [-2.0, 3.0], [3.0, 4.0], [1.2, -0.5],[-3.2,4.5],[0.02, 3.5],
[1.0, 2.0],[4,12],[10,-2],[3,14],[2.7,6],[5,1],[-4,2],[6,-2],[-0.5,2],[2,8],[3,0],[2,1],[-2,2],
[-1.5,5],[2,4.7],[3.8,10],[0.5,3],[0.6,1.8],[2,3],[0.2,1.5],[-0.5,1],[-0.8,0.8],[-1,2.1]])

y_data = torch.tensor([0,2,1,0,2,  2,1,1,0,1,  1,0,2,0,2,1, 0,0,2,2,1, 1,2,1,1,1,1,1,2])
```

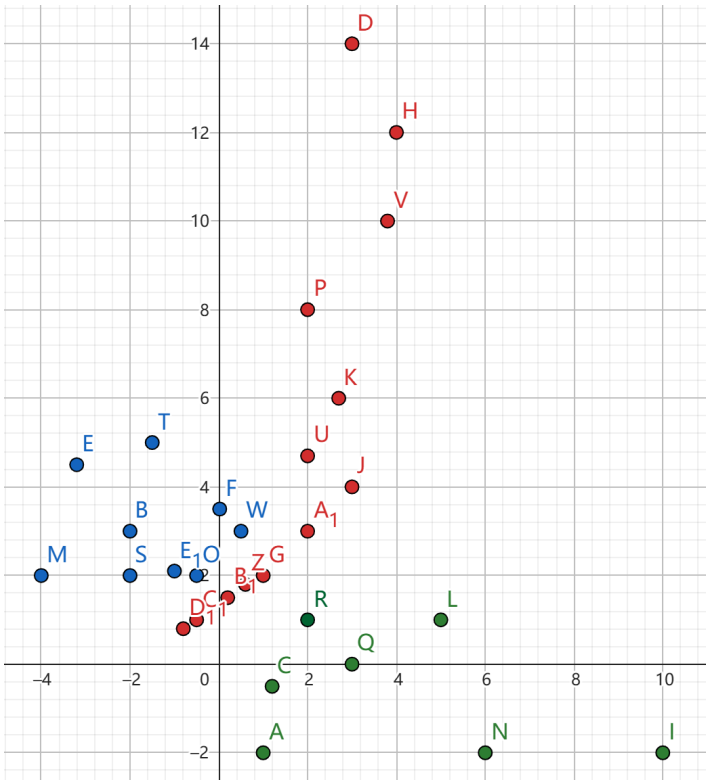

Figure 2: 代码中的 data

而测试样本点则是这 10 个点：

```
testdata_x = torch.tensor([[1.0, -1.0], [-2.0, 1.2], [3.0, 10], [1.2, 0],[-1,2], [1.0, -1.5],
[-1.7, 0.5], [3, 3], [1, 0.1],[-1,2.4]])

testdata_y = torch.tensor([0,2,1,0,2,0,1,1,0,2])
```

设置本数据集原因在于：红色类是一个倾斜的狭窄的区域，插入到了蓝色和绿色两类中间，这给分类会带来一定挑战，使用普通的 KNN 算法很难处理红蓝绿交汇地带数据点的分类，如果经过 Adaboost 提升的 KNN 可以达到较好的分类效果，说明得到了一个较强的分类器，本方法有效。

Results

`for cnt in range(0,5)`迭代 5 次时的输出：（这几个 error tensor 分别代表这几轮更新时的犯错率，打印出的一长串张量是我的模型，最后底部是最终模型的效果）

```
error tensor(0.2500)
error tensor(0.7778)
error tensor(0.5000)
error tensor(0.2500)
error tensor(0.6667)

 Here is my classifier

x_train: tensor([[ 2.0000,  3.0000],
        [ 1.2000, -0.5000],
        [ 3.0000,  0.0000],
        [ 0.6000,  1.8000],
        [ 3.0000, 14.0000],
        [10.0000, -2.0000],
        [ 0.2000,  1.5000],
        [ 2.0000,  1.0000],
```

```
        [-2.0000,  3.0000],
        [ 2.0000,  8.0000],
        [ 5.0000,  1.0000],
        [ 1.0000, -2.0000],
        [-2.0000,  2.0000],
        [ 3.0000,  4.0000],
        [-1.5000,  5.0000],
        [-1.0000,  2.1000],
        [-3.2000,  4.5000],
        [ 1.0000,  2.0000],
        [ 0.5000,  3.0000],
        [-4.0000,  2.0000],
        [ 0.0200,  3.5000],
        [ 2.0000,  4.7000],
        [-0.5000,  1.0000],
        [ 3.8000, 10.0000],
        [ 4.0000, 12.0000]])
y_train: tensor([1, 0, 0, 1, 1, 0, 1, 0, 2, 1, 0, 0, 2, 1, 2, 2, 2, 1, 2, 2, 2, 1, 1, 1,
        1])
w_train: tensor([0.0294, 0.0294, 0.0294, 0.0294, 0.0294, 0.0294, 0.0294, 0.0294, 0.0294,
        0.0294, 0.0294, 0.0294, 0.0294, 0.0294, 0.0294, 0.0294, 0.0294, 0.0294,
        0.0294, 0.0294, 0.0294, 0.0294, 0.0294, 0.0294, 0.0294, 0.0294, 0.0294])
alpha: 1.791759469228055
x_train: tensor([[ 0.2000,  1.5000],
        [10.0000, -2.0000],
        [ 2.0000,  3.0000],
        [ 2.7000,  6.0000],
        [-2.0000,  3.0000],
        [ 0.6000,  1.8000],
        [ 3.0000,  0.0000],
        [-0.5000,  1.0000],
        [ 3.8000, 10.0000],
        [-0.8000,  0.8000],
        [ 0.5000,  3.0000],
        [ 1.0000, -2.0000],
        [-1.5000,  5.0000],
        [ 3.0000,  4.0000],
        [-3.2000,  4.5000],
        [ 2.0000,  8.0000],
        [-1.0000,  2.1000],
        [ 1.2000, -0.5000],
        [ 4.0000, 12.0000],
        [-2.0000,  2.0000],
        [ 5.0000,  1.0000],
        [ 3.0000, 14.0000],
        [ 0.0200,  3.5000],
        [ 2.0000,  4.7000],
        [ 1.0000,  2.0000]])
y_train: tensor([1, 0, 1, 1, 2, 1, 0, 1, 1, 1, 2, 0, 2, 1, 2, 1, 2, 0, 1, 2, 0, 1, 2, 1,
        1])
w_train: tensor([0.0323, 0.0323, 0.0323, 0.0323, 0.0323, 0.0323, 0.0323, 0.0323, 0.0323,
        0.0323, 0.0323, 0.0323, 0.0323, 0.0323, 0.0323, 0.0323, 0.0323, 0.0323,
        0.0323, 0.0323, 0.0323, 0.0323, 0.0323, 0.0323, 0.0323])
alpha: -0.5596155346157144
x_train: tensor([[ 0.6000,  1.8000],
        [ 4.0000, 12.0000],
```

```
        [-0.8000,  0.8000],
        [ 0.0200,  3.5000],
        [-4.0000,  2.0000],
        [ 3.0000, 14.0000],
        [ 2.0000,  8.0000],
        [-1.5000,  5.0000],
        [ 2.0000,  4.7000],
        [ 3.0000,  0.0000],
        [ 1.0000, -2.0000],
        [ 2.0000,  1.0000],
        [-3.2000,  4.5000],
        [ 5.0000,  1.0000],
        [ 6.0000, -2.0000],
        [ 1.0000,  2.0000],
        [-1.0000,  2.1000],
        [ 3.8000, 10.0000],
        [-0.5000,  1.0000],
        [-2.0000,  2.0000],
        [10.0000, -2.0000],
        [ 3.0000,  4.0000],
        [ 2.0000,  3.0000],
        [-0.5000,  2.0000],
        [ 2.7000,  6.0000]])
y_train: tensor([1, 1, 1, 2, 2, 1, 1, 2, 1, 0, 0, 0, 2, 0, 0, 1, 2, 1, 1, 2, 0, 1, 1, 2,
        1])
w_train: tensor([0.0303, 0.0303, 0.0303, 0.0303, 0.0303, 0.0303, 0.0303, 0.0303, 0.0303,
        0.0303, 0.0303, 0.0173, 0.0303, 0.0303, 0.0303, 0.0303, 0.0303, 0.0303,
        0.0303, 0.0303, 0.0303, 0.0303, 0.0303, 0.1039, 0.0303])
alpha: 0.6931471805599453
x_train: tensor([[-1.0000,  2.1000],
        [-4.0000,  2.0000],
        [-3.2000,  4.5000],
        [ 5.0000,  1.0000],
        [-2.0000,  2.0000],
        [ 3.0000,  4.0000],
        [ 2.0000,  3.0000],
        [ 2.0000,  4.7000],
        [ 2.0000,  8.0000],
        [ 2.0000,  1.0000],
        [ 3.0000, 14.0000],
        [ 1.2000, -0.5000],
        [ 0.5000,  3.0000],
        [-0.5000,  2.0000],
        [ 2.7000,  6.0000],
        [ 6.0000, -2.0000],
        [ 0.2000,  1.5000],
        [ 1.0000,  2.0000],
        [-0.8000,  0.8000],
        [ 1.0000, -2.0000],
        [ 0.0200,  3.5000],
        [ 3.8000, 10.0000],
        [ 3.0000,  0.0000],
        [-1.5000,  5.0000],
        [-0.5000,  1.0000]])
y_train: tensor([2, 2, 2, 0, 2, 1, 1, 1, 1, 0, 1, 0, 2, 2, 1, 0, 1, 1, 1, 0, 2, 1, 0, 2,
        1])
```

```
w_train: tensor([0.0263, 0.0263, 0.0263, 0.0263, 0.0263, 0.0263, 0.0263, 0.0263, 0.0263,
        0.0150, 0.0263, 0.0263, 0.0526, 0.0902, 0.0263, 0.0263, 0.0526, 0.0263,
        0.0263, 0.0263, 0.0263, 0.0263, 0.0263, 0.0263, 0.0263])
alpha: 1.791759469228055
x_train: tensor([[ 1.0000,  2.0000],
        [ 0.0200,  3.5000],
        [-1.0000,  2.1000],
        [ 6.0000, -2.0000],
        [ 0.2000,  1.5000],
        [ 0.5000,  3.0000],
        [-0.5000,  1.0000],
        [ 1.2000, -0.5000],
        [-1.5000,  5.0000],
        [-2.0000,  2.0000],
        [ 5.0000,  1.0000],
        [ 3.0000, 14.0000],
        [ 2.0000,  8.0000],
        [-3.2000,  4.5000],
        [10.0000, -2.0000],
        [-0.5000,  2.0000],
        [ 3.0000,  0.0000],
        [ 3.8000, 10.0000],
        [-4.0000,  2.0000],
        [-0.8000,  0.8000],
        [ 2.0000,  1.0000],
        [ 4.0000, 12.0000],
        [ 3.0000,  4.0000],
        [ 2.0000,  3.0000],
        [-2.0000,  3.0000]])
y_train: tensor([1, 2, 2, 0, 1, 2, 1, 0, 2, 2, 0, 1, 1, 2, 0, 2, 0, 1, 2, 1, 0, 1, 1, 1,
        2])
w_train: tensor([0.0263, 0.0263, 0.0263, 0.0263, 0.0526, 0.0526, 0.0263, 0.0263, 0.0263,
        0.0263, 0.0263, 0.0263, 0.0263, 0.0263, 0.0263, 0.0902, 0.0263, 0.0263,
        0.0263, 0.0263, 0.0150, 0.0263, 0.0263, 0.0263, 0.0263])
alpha: -5.960464655174746e-08

----------in the original data set, the prediction result is---------
2
error rate 0.06896551724137931
----------now lets test it with new input dots, the prediction result is---------
1
error rate 0.1
```

更改循环次数，cnt 设置为 `for cnt in range(0,10)`以及 `for cnt in range(0,20)`

```
y_train: tensor([2, 2, 1, 1, 1, 0, 2, 2, 0, 1, 1, 1, 0, 1, 1, 0, 1, 2, 1, 1, 0, 1, 2, 0,
        0])
w_train: tensor([0.0267, 0.0067, 0.0267, 0.0267, 0.0267, 0.0267, 0.0067, 0.0267, 0.0267,
        0.0267, 0.0267, 0.0267, 0.0267, 0.0267, 0.0267, 0.0267, 0.0267, 0.0267,
        0.0267, 0.0267, 0.0267, 0.0267, 0.1600, 0.0267, 0.1600])
alpha: 1000
----------in the original data set, the prediction result is---------
1
error rate 0.034482758620689655
----------now lets test it with new input dots, the prediction result is---------
0
error rate 0.0
```

Figure 3: 迭代 10 次后结果

```
          [ 0.2000,  1.5000],
          [ 1.0000, -2.0000],
          [ 1.2000, -0.5000],
          [ 3.8000, 10.0000],
          [ 2.7000,  6.0000],
          [ 3.0000,  4.0000]])
y_train: tensor([2, 2, 1, 2, 1, 2, 1, 0, 2, 2, 1, 1, 1, 1, 2, 0, 0, 0, 1, 1, 0, 0, 1, 1,
        1])
w_train: tensor([0.0861, 0.0236, 0.0236, 0.0236, 0.0236, 0.0082, 0.0808, 0.0236, 0.0236,
        0.0236, 0.0629, 0.0236, 0.0314, 0.0236, 0.0236, 0.0236, 0.0236, 0.0236,
        0.0629, 0.0236, 0.0236, 0.0236, 0.0236, 0.0236, 0.0314])
alpha: -5.960464655174746e-08
----------in the original data set, the prediction result is---------
0
error rate 0.0
----------now lets test it with new input dots, the prediction result is---------
0
error rate 0.0
```

Figure 4: 迭代 20 次后结果

可见，当迭代 5 次时，在原始样本集合上就有 0.069 的错误率，而在测试样本上有 0.1 的错误率；当迭代 10 次时，在原始样本上有 0.0345 的错误率，而在测试样本上全部分类正确；当迭代 20 次时，在原始训练集上达到完全正确分类，在测试样本上也完全正确，这说明本方法有效。

下面在汇集区域又增加了一些点，如下是新的数据集：

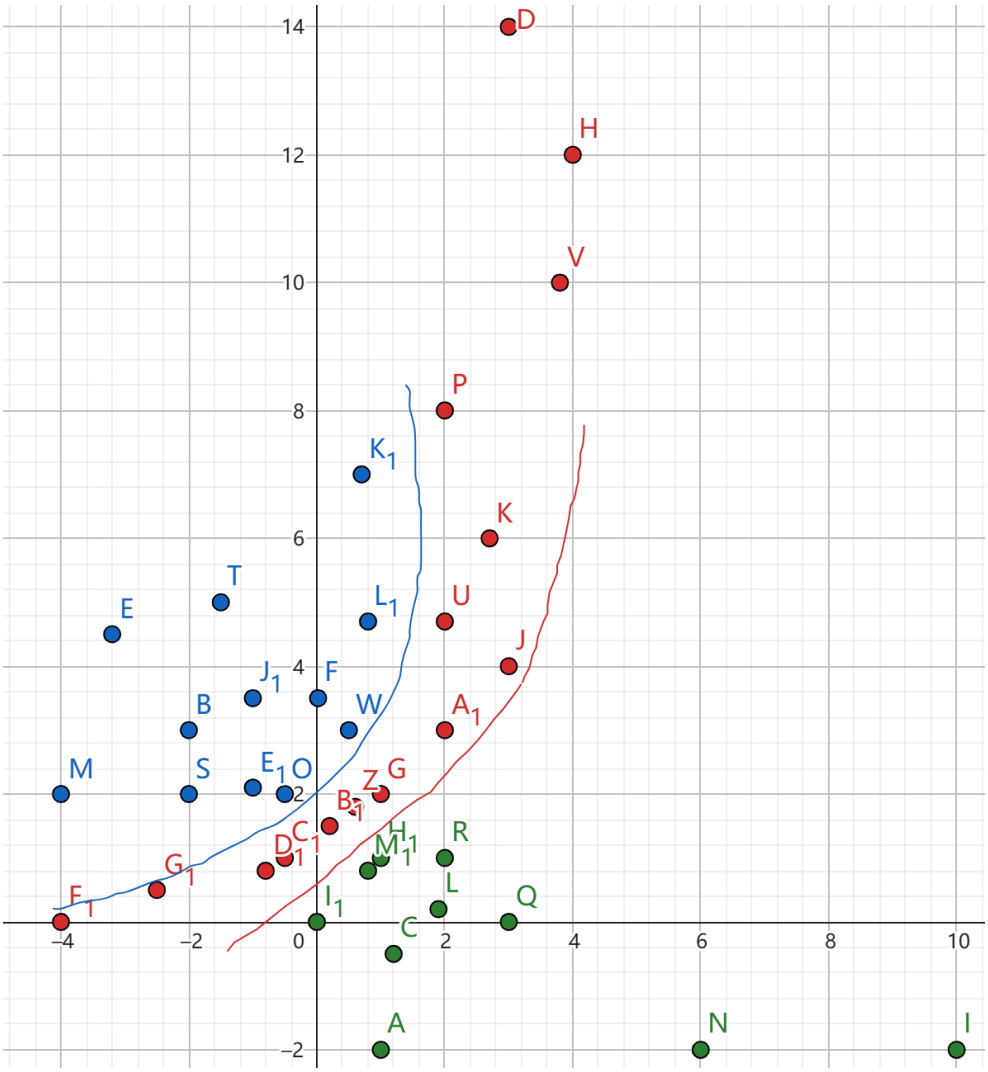

Figure 5: 增多点数，新的数据集

其测试效果为：

迭代次数	训练集上的错误率	测试集上的错误率
3	0.0526	0.1
5	0.1053	0.1
10	0.0526	0.1
20	0.0789	0
30	0.0263	0.1
50	0.0263	0.1
100	0.0526	0.1
200	0.0526	0.1
500	0.1053	0.1

可见，还是有缺点，此时迭代次数的增大也不能提高正确率了（甚至会更糟糕），一直有几个“钉子户”无法被正确归类

更改 K 近邻的 K 值为 5 后，即上述代码中取 knn(5)，结果在原始数据集上表现更加糟糕，取 knn(7)，比 5 更加糟糕

Summary

由于数据集需要 DIY，因此数据集还是偏小了，如果数据集更多会更能体现模型的优劣

就目前情况来看，根据这种方法迭代 20 次左右已经可以达到较好的预测效果，不需要更大的迭代次数几乎是没有意义的

此外，knn 中 k 值的选取也很重要，根据调参结果，取 k 为 3 左右会有较好的效果（因为我的样本点数也较少），k 大概是总的点数的 8% 左右较好