支持向量机实验报告

姓名：_____ 张韫译萱_____　学号：_____08023214_____

一、 实验题目

1 数据集

实验使用 DNA 数据集，包含：

- 3 个类别
- 180 个属性（2 个属性值）
- 1400 个训练样本（dna.scale.tr）
- 600 个验证样本（dna.scale.val）
- 1186 个测试样本（dna.scale.t）

文件格式：纯文本

注：验证集用于调试超参数

2 实验任务

1. 熟悉数据集
2. 运行 libsvm 算法

3 实验要求

1. Python 实现
2. 分别使用线性核、多项式核、高斯核等，记录并比较准确率

二、 实验原理

1 支持向量机概述

支持向量机（Support Vector Machine, SVM）是一种基于统计学习理论的监督学习方法，由 Vapnik 等人于 1995 年提出。SVM 的核心思想是在特征空间中寻找一个最优分离超平面，使得不同类别样本之间的间隔最大化，从而获得更好的泛化能力。

2 线性支持向量机

对于线性可分的二分类问题，给定训练数据集 $\{(x_i, y_i)\}_{i=1}^n$，其中 $x_i \in \mathbf{R}^d$ 为特征向量，$y_i \in \{-1, +1\}$ 为类别标签。支持向量机的目标是找到一个分离超平面：

$$w^T x + b = 0$$

使得所有训练样本满足约束条件：

$$y_i(w^T x_i + b) \geq 1, \quad i = 1, 2, \ldots, n$$

同时最大化分类间隔 $\frac{2}{\|w\|}$。等价于求解以下优化问题：

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

$$\text{s.t.} \quad y_i(w^T x_i + b) \geq 1, \quad i = 1, 2, \ldots, n$$

通过引入拉格朗日乘子 $\alpha_i \geq 0$，得到对偶问题：

$$\max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2}\sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j x_i^T x_j$$

$$\text{s.t.} \quad \sum_{i=1}^n \alpha_i y_i = 0, \quad \alpha_i \geq 0, \quad i = 1, 2, \ldots, n$$

对于线性不可分的情况，我们引入松弛变量 ξ_i 和惩罚参数 C，得到软间隔支持向量机：

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^n \xi_i$$

$$\text{s.t.} \quad y_i(w^T x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0, \quad i = 1, 2, \ldots, n$$

其中参数 C 控制对误分类样本的惩罚程度。C 值越大，对误分类的惩罚越重，模型可能过拟合；C 值越小，模型泛化能力越强，但可能欠拟合。

3 核函数方法

对于非线性问题，SVM 通过核函数 $K(x_i, x_j)$ 将原始特征空间映射到高维特征空间，使得样本在高维空间中线性可分。核函数定义为：

$$K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$$

其中 $\phi(\cdot)$ 是从原始空间到高维空间的映射函数。通过引入核函数，可以在不显式计算映射的情况下完成高维空间中的内积运算。对偶问题变为：

$$\max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2}\sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j K(x_i, x_j)$$

4 常用核函数

4.1 线性核

线性核函数的定义为：

$$K(x_i, x_j) = x_i^T x_j$$

线性核适用于线性可分或近似线性可分的数据集，具有计算简单、训练速度快的优点。

4.2 多项式核

多项式核函数的定义为：

$$K(x_i, x_j) = (\gamma x_i^T x_j + r)^d$$

其中 γ 是核系数，r 是常数项，d 是多项式的度数。多项式核可以学习样本间的高阶关系，适合处理非线性问题。

4.3 高斯核（RBF 核）

高斯核也称为径向基函数核，定义为：

$$K(x_i, x_j) = \exp\left(-\gamma \|x_i - x_j\|^2\right)$$

其中 γ 控制核函数的宽度。γ 值越大，核函数影响范围越小，模型越复杂；γ 值越小，影响范围越大，模型越平滑。高斯核将样本映射到无限维空间，具有很强的非线性学习能力，是应用最广泛的核函数之一。

5 多分类策略

对于 k 类分类问题，libsvm 采用” 一对一”（One-vs-One）策略，构建 $\frac{k(k-1)}{2}$ 个二分类器。对于 DNA 数据集的 3 个类别，需要构建 3 个二分类器。在预测时，采用投票机制：每个二分类器对测试样本进行预测，获得票数最多的类别作为最终预测结果。

6 参数优化

SVM 的性能很大程度上依赖于参数的选择。主要参数包括：

- **惩罚参数** C：控制对误分类样本的惩罚程度
- **核参数** γ：控制单个训练样本的影响范围

参数优化通常采用网格搜索（Grid Search）方法：在给定的参数空间内，以一定的步长遍历所有参数组合，通过验证集评估每组参数的性能，选择验证集准确率最高的参数组合作为最优参数。

三、 过程与结果

1 实验流程

1. 用 `svm_read_problem` 函数读取训练集和测试集数据
2. 使用 `svm_problem` 函数构建 SVM 问题实例
3. 设置不同的核函数及参数
4. 调用 `svm_train` 函数训练 SVM 模型
5. 使用 `svm_predict` 函数分别在训练集和测试集上评估模型性能，对比性能差异

2 参数设置

在这里，为三种核函数选择以下参数组合：

- **线性核**：`-t 0 -c 0.03125 -g 0.0078125`
- **多项式核**：`-t 1 -c 16 -g 1`
- **高斯核**：`-t 2 -c 2 -g 0.0078125`

其中，`-t` 参数指定核函数类型（0 表示线性核，1 表示多项式核，2 表示高斯核），`-c` 参数指定惩罚系数 C，`-g` 参数指定核函数参数 γ。

3 选取线性核的正确率

使用线性核函数进行训练，参数设置为 $C = 0.03125$，$\gamma = 0.0078125$。实验结果如下：

- **训练集准确率**：97.29%（1362/1400）
- **测试集准确率**：96.65%（1933/2000）

线性核在 DNA 数据集上取得了较好的分类效果。训练集和测试集准确率相近，准确率差异为 0.64%，表明模型具有良好的泛化能力，未出现明显的过拟合现象。线性核的优势在于计算简单、训练速度快，适合作为基线方法。

4 选取多项式核的正确率

使用多项式核函数进行训练，参数设置为 $C = 16$，$\gamma = 1$。实验结果如下：

- **训练集准确率**：100.00%（1400/1400）
- **测试集准确率**：98.30%（1966/2000）

多项式核在训练集上达到了完美分类，测试集准确率也达到了三种核函数中的最高值。这说明多项式核能够有效捕捉 DNA 数据中的高阶特征关系。训练集准确率为 100% 而测试集为 98.30%，准确率差异为 1.70%，存在轻微的过拟合倾向，但测试集性能仍然优异。多项式核通过学习特征间的高阶交互，在本数据集上表现最优。

5 选取高斯核的正确率

使用高斯核函数进行训练，参数设置为 $C = 2$，$\gamma = 0.0078125$。实验结果如下：

- **训练集准确率**：98.21%（1375/1400）
- **测试集准确率**：97.60%（1952/2000）

高斯核的性能介于线性核和多项式核之间，在训练集和测试集上都表现出良好的分类能力。训练集和测试集准确率差异仅为 0.61%，模型泛化性能良好。高斯核将样本映射到无限维空间，具有强大的非线性学习能力，同时保持了较好的泛化性能。

6 实验结果对比

图4展示了三种核函数在训练集和测试集上的分类准确率对比：

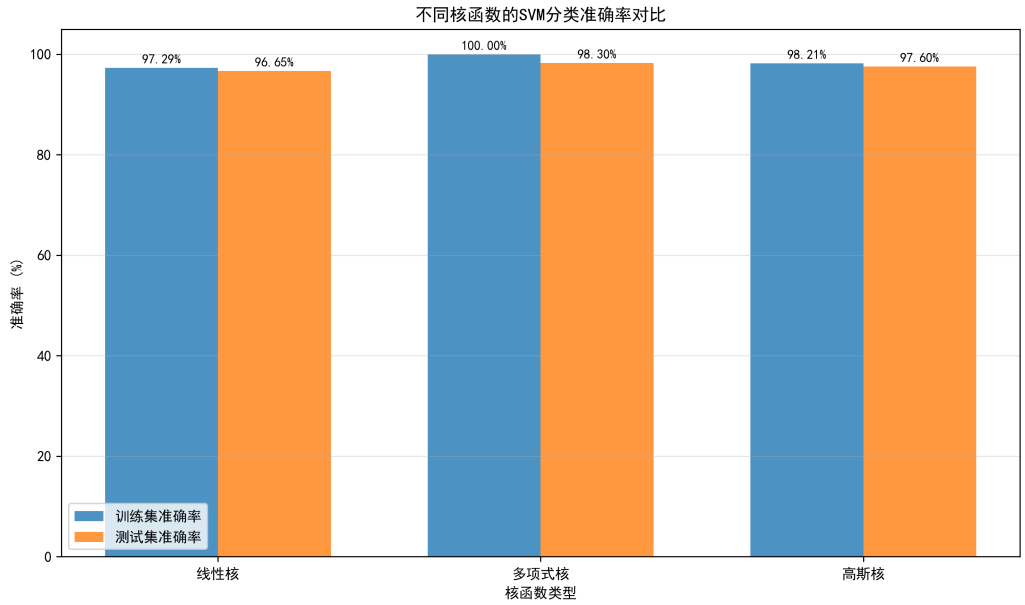

图 1: 不同核函数的 SVM 分类准确率对比

从实验结果可以看出：

- 多项式核在测试集上取得了最高的准确率（98.30%），表现最优
- 高斯核次之（97.60%），在训练集和测试集上表现均衡
- 线性核表现相对较弱（96.65%），但仍达到了较高的准确率
- 所有核函数的测试集准确率都在 96% 以上，说明 DNA 数据集的分类任务完成度较高

通过对比三种核函数的实验结果，可以看出：

6.1 核函数选择的影响

不同核函数适用于不同的数据特性。对于 DNA 数据集，线性核针对处理线性问题，在数据集上的性能略低。多项式核能够学习特征间的高阶交互关系，在数据集上表现最优。高斯核具有强大的非线性学习能力，性能稳定，泛化能力强。（在问题类型未知时，应预先选择高斯核进行处理）。

6.2 过拟合分析

从训练集和测试集准确率的差异可以分析模型的拟合情况。多项式核训练集准确率 100%，测试集 98.30%，差异 1.70%，存在轻微过拟合。高斯核训练集准确率 98.21%，测试集 97.60%，差异 0.61%，拟合良好。线性核训练集准确率 97.29%，测试集 96.65%，差异 0.64%，拟合良好。

6.3 支持向量数量

从训练过程输出可以看出，不同核函数对应的支持向量数量不同。线性核的支持向量数为 574，多项式核的支持向量数为 1124，高斯核的支持向量数为 701。支持向量数量反映了模型的复杂度，数量越多说明数据的非线性程度越高。

libsvm 使用简便，单步易于完整理解使用 SVM 求解问题的理论推导过程。我使用 Pytorch 复现了 libsvm 中基于核方法，通过求解软约束优化问题实现分类的部分功能，在西瓜数据集上使用线性核，多项式核和高斯核对 SVM 的性能进行了测试，发现在训练集上（西瓜数据集并没有划分测试集），使用二次多项式核，分类准确率达到 100%，使用线性核，分类准确率为 64.71%，使用高斯核，分类准确率为 82.35%，和在 DNA 数据集上得到的结果有相似性。

7 Pytorch 代码测试结果

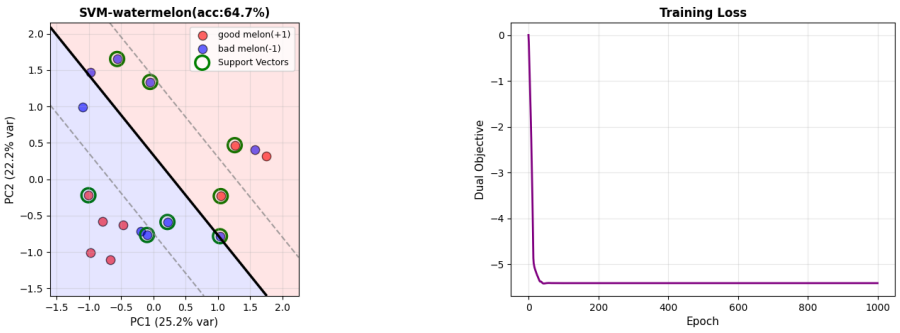

图 2: 使用线性核的训练结果（分类结果进行了降维处理）

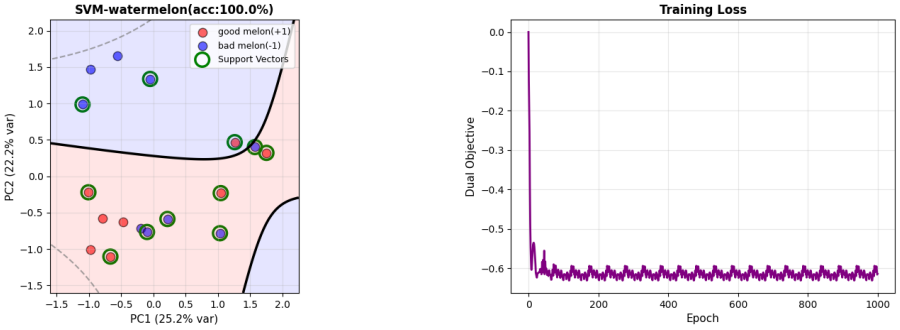

图 3: 使用二次多项式核的训练结果（分类结果进行了降维处理）

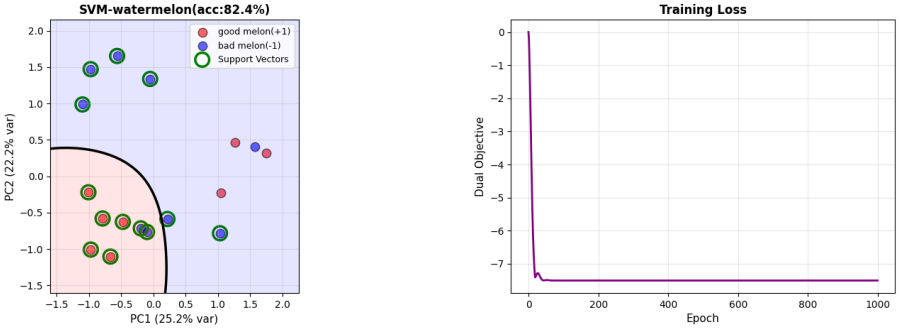

图 4: 使用高斯核的训练结果（分类结果进行了降维处理）

四、 实验总结

通过本次实验，我学习了支持向量机的基本原理和应用方法，掌握了 libsvm 工具的使用，并在 DNA 数据集上完成了分类任务，通过实际操作加深了对 SVM 理论的理解，验证了课堂上学习的理论知识。

五、 代码

1 main.py

```
from libsvm.svmutil import *
from libsvm.svm import *
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rcParams['font.sans-serif'] = ['SimHei']
matplotlib.rcParams['axes.unicode_minus'] = False

print("正在加载数据...")
y, x = svm_read_problem('dna.scale.tr.txt')
yv, xv = svm_read_problem('dna.scale.val.txt')
```

```
yt, xt = svm_read_problem('dna.scale.txt')
prob = svm_problem(y, x)

kernel_params = [
    ('-t 0 -c 0.03125 -g 0.0078125', '线性核'),
    ('-t 1 -c 16 -g 1', '多项式核'),
    ('-t 2 -c 2 -g 0.0078125', '高斯核')
]

results = []
for params, kernel_name in kernel_params:
    print(f'\n========== {kernel_name} ==========')
    param = svm_parameter(params)
    model = svm_train(prob, param)
    svm_save_model(f'model_svm_{kernel_name}', model)
    print(f'{kernel_name} - Train:')
    p_label, p_acc, p_val = svm_predict(y, x, model)
    train_acc = p_acc[0]
    print(f'{kernel_name} - Test:')
    p_label, p_acc, p_val = svm_predict(yt, xt, model)
    test_acc = p_acc[0]
    results.append((kernel_name, train_acc, test_acc))

print('\n========== 实验结果汇总 ==========')
for kernel_name, train_acc, test_acc in results:
    print(f'{kernel_name}: 训练集准确率={train_acc:.2f}%, '
          f'测试集准确率={test_acc:.2f}%')

kernels = [r[0] for r in results]
train_accs = [r[1] for r in results]
test_accs = [r[2] for r in results]

fig, ax = plt.subplots(figsize=(10, 6))
x_pos = range(len(kernels))
width = 0.35
ax.bar([p - width/2 for p in x_pos], train_accs, width,
       label='训练集准确率', alpha=0.8)
ax.bar([p + width/2 for p in x_pos], test_accs, width,
       label='测试集准确率', alpha=0.8)
ax.set_xlabel('核函数类型')
ax.set_ylabel('准确率 (%)')
ax.set_title('不同核函数的SVM分类准确率对比')
ax.set_xticks(x_pos)
ax.set_xticklabels(kernels)
ax.legend()
ax.grid(axis='y', alpha=0.3)
for i, (train, test) in enumerate(zip(train_accs, test_accs)):
    ax.text(i - width/2, train + 0.5, f'{train:.2f}%',
```

```
            ha='center', va='bottom', fontsize=9)
    ax.text(i + width/2, test + 0.5, f'{test:.2f}%',
            ha='center', va='bottom', fontsize=9)
plt.tight_layout()
plt.savefig('svm_results.png', dpi=300, bbox_inches='tight')
print('\n结果可视化图表已保存为 svm_results.png')
plt.show()
```

2 Torch_SVM.py

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.decomposition import PCA

class SVMNetwork_Original(nn.Module):#原始的软约束SVM，使用hinge损失函数
    """原始线性SVM（保留用于对比）"""
    def __init__(self,in_shape):
        super(SVMNetwork_Original,self).__init__()
        self.linear = nn.Linear(in_shape,1)

    def forward(self,x):
        return self.linear(x)

    def calloss(self,x,y,c):#x:(batch_size,in_shape) y:(batch_size,1)
        L2_item = 0.5 * self.linear.weight.pow(2).sum()
        hinge_loss = c*(torch.sum(torch.clamp(1 - y*self.forward(x), min=0)))
        return (L2_item + hinge_loss)

    def calloss_kernel(self,x,y,c):
        L2_item = 0.5 * self.linear.weight.pow(2).sum()


    def optimiser(self,lr,decay,momentum=None):
        if momentum != None:
            return optim.SGD(self.parameters(), lr=lr,weight_decay=decay,momentum=momentum)
        else:
            return optim.AdamW(self.parameters(), lr=lr,weight_decay=decay)

    def trainmodel(self,epochs,x_train,y_train,x_val,y_val,c=0.005,lr=0.01,decay=0.0005,
    momentum=0.9): #c不能设的太高
        #深复制一份，然后断开计算图（如果x,y进来的时候带着梯度，不这样做就会出问题，会更新它们的
    梯度）
        x_train = x_train.clone().detach()
```

```
        y_train = y_train.clone().detach().view(-1, 1)
        x_val = x_val.clone().detach()
        y_val = y_val.clone().detach().view(-1, 1)
        opt = self.optimiser(lr,decay,momentum)
        train_loss = []
        val_loss = []
        for i in range(epochs):
            self.train()
            opt.zero_grad()
            loss = self.calloss(x_train,y_train,c)
            loss.backward()
            opt.step()
            train_loss.append(loss.item())

            self.eval()
            with torch.no_grad():
                loss = self.calloss(x_val,y_val,c)
                val_loss.append(loss.item())
            print(f"epoch {i},train_loss:{train_loss[-1]},val_loss:{val_loss[-1]}")
        return train_loss,val_loss

    def visualize_svm(self, X, y, losses):
        """可视化SVM决策边界和训练过程"""
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

        # 左图：决策边界
        X_np = X.numpy()
        y_np = y.numpy().flatten()

        # 绘制数据点
        ax1.scatter(X_np[y_np == 1, 0], X_np[y_np == 1, 1],
                    c='red', label='Class +1', alpha=0.6, edgecolors='k')
        ax1.scatter(X_np[y_np == -1, 0], X_np[y_np == -1, 1],
                    c='blue', label='Class -1', alpha=0.6, edgecolors='k')

        # 绘制决策边界
        w = self.linear.weight.data.numpy().flatten()
        b = self.linear.bias.data.numpy()[0]

        # 决策边界：w1*x1 + w2*x2 + b = 0
        x1_min, x1_max = X_np[:, 0].min() - 1, X_np[:, 0].max() + 1
        x1 = np.linspace(x1_min, x1_max, 100)
        x2 = -(w[0] * x1 + b) / w[1]

        ax1.plot(x1, x2, 'k-', linewidth=2, label='Decision Boundary')

        # 绘制支持向量边界 (margin)
        x2_margin_plus = -(w[0] * x1 + b - 1) / w[1]
```

```
84          x2_margin_minus = -(w[0] * x1 + b + 1) / w[1]
85          ax1.plot(x1, x2_margin_plus, 'k--', linewidth=1, alpha=0.5)
86          ax1.plot(x1, x2_margin_minus, 'k--', linewidth=1, alpha=0.5)
87
88          ax1.set_xlabel('Feature 1')
89          ax1.set_ylabel('Feature 2')
90          ax1.set_title('SVM Decision Boundary')
91          ax1.legend()
92          ax1.grid(True, alpha=0.3)
93
94          # 右图：损失曲线
95          ax2.plot(losses, linewidth=2)
96          ax2.set_xlabel('Epoch')
97          ax2.set_ylabel('Loss')
98          ax2.set_title('Training Loss')
99          ax2.grid(True, alpha=0.3)
100
101         plt.tight_layout()
102         plt.show()
103
104 class KernelSVM(nn.Module):#基于对偶方法的SVM，通过选择合适的核函数可以做高维空间的二分类问题
105     def __init__(self,input_dim=1,sigma=None,C=None,d=None):
106         super(KernelSVM,self).__init__()
107         self.sigma = sigma #高斯核
108         self.C = C #软间隔 batch batch
109         self.d = d #多项式核
110         self.alpha = None #alpha
111         self.b = None #基于支持向量计算
112
113     def kernel(self,x1,x2):#逐个对每两个样本计算核函数（结果是(batch,batch)）
114         if self.d!=None:#(batch,dim)
115             return torch.mm(x1,x2.t()) ** self.d #要的就是逐元素幂，不是矩阵乘法
116         elif self.sigma!=None:#x1**2+x2**2-2x1x2()
117             dist2 = torch.cdist(x1, x2, p=2) ** 2  # 欧氏距离平方 (batch1, batch2)
118             # mask = ~torch.eye(dist2.size(0), dtype=torch.bool)
119             # off_diag_elements = dist2[mask]
120             # # 求平均
121             # mean_value = off_diag_elements.mean()
122             self.sigma = 1 #sigma不能变化，不然就丢失核本身的特性了
123             return torch.exp(-dist2/(2*self.sigma**2))
124             #return torch.exp(-(torch.sum(x1**2,dim=1)+torch.sum(x2**2,dim=1)-2*torch.mm(x1,
    x2.t()))/(2*self.sigma**2))#(batch,1)+(batch,1)+(batch,batch)
125         else:
126             return torch.mm(x1,x2.t())
127
128     def calloss(self,x,y):
129         #标准流程
130         x = x.clone().detach()
```

```
        y = y.clone().detach().view(-1,1)#避免传进来的是个向量造成运算错误
        # return 0.5 * self.alpha.t() @ y.t() @ self.kernel(x,x) @ y @ self.alpha
        yky = torch.mm(y,y.t())*self.kernel(x,x) #先yTy做矩阵乘法，再逐元素与核矩阵相乘（技巧是只要维度匹配，不丢项，得到的就是唯一正确的解）
        return (0.5 * torch.mm(torch.mm(self.alpha.t(),yky),self.alpha) - torch.sum(self.alpha,dim=0)) #这里是alphaT@yky@alpha

    def trainmodel(self,x_train,y_train,epochs=1000):
        x_train = x_train.clone().detach()
        y_train = y_train.clone().detach().view(-1, 1)
        self.alpha = nn.Parameter(torch.zeros(x_train.size()[0],1,requires_grad=True)) #alpha是拉格朗日乘子！数量是样本数！！！
        self.train()
        optimiser = optim.SGD(params=self.parameters(),lr=0.01,momentum=0.9,weight_decay=0) #优化\alpha
        train_loss = []
        for i in range(epochs):
            #更新参数
            self.train()
            optimiser.zero_grad()
            loss = self.calloss(x_train,y_train)
            loss.backward()
            optimiser.step()
            train_loss.append(loss.item())

            self.eval()
            with torch.no_grad():
                #首先引入软约束条件，强制满足约束（alpha \of[0,C]）
                self.alpha.clamp_(0,self.C)
                sum_y = torch.sum(self.alpha*y_train) #\sigma alpha_i yi =0
                if(torch.abs(sum_y)>=0.001):
                    alpha_index = (self.alpha>1e-5) & (self.alpha<self.C-1e-5)
                    """ (alpha-corrlation * y)*y = alpha *y - corrlation * mask.sum() = 0, corrlation=sum/mask.sum(),
                    mask.sum()会返回所有需要调整的alpha的个数（True+1,false+0） """
                    corrlation = sum_y / alpha_index.sum()
                    self.alpha[alpha_index] -= corrlation * y_train[alpha_index]
                    self.alpha.clamp_(0,self.C)
                #寻找支持向量，阈值为1e-4，离边界至少这个距离，在边界范围之内的为支持向量
                support_index = ((self.alpha.view(-1) > 1e-5) & (self.alpha.view(-1) < self.C - 1e-5)).nonzero(as_tuple=True)[0] #记得要转成向量，一维的，方便推导;nonzero会返回结果是True的元组，每个里面是一个向量，取[0]把里面的向量(tensor，可以直接用于提取下标),and，or链式比较操作符不能直接用于多元素张量，要用&
                support_vec = x_train[support_index]
                support_label = y_train[support_index]
                support_alpha = self.alpha[support_index]
                self.support_vec = support_vec
                self.support_label = support_label
```

```
171                 self.support_alpha = support_alpha
172                 #计算b,b=\frac{1}{|S|} \sum_{s \in S}\left(y_s-\sum_{i \in S} \alpha_i y_i \
    boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{x}_s\right) .
173                 self.b = torch.mean(support_label - torch.sum(support_alpha*support_label*
    self.kernel(support_vec,support_vec),dim=1))#(len(lndex),len(index))->sum(len(index))
174             #print(f"epoch {i},train_loss:{loss.item()},bia:{self.b.item():.4f}, alpha_min:{
    self.alpha.min().item():.4f}, alpha_max:{self.alpha.max().item():.4f}, sum_y:{sum_y.
    item():.4f}")
175         return train_loss
176 
177     def decision_function(self,x):#\sigma \alpha_i y_ik(x,x_i) + b,y_i,x_i都是从样本中挑出来
    的的支持向量
178         kernel = self.kernel(self.support_vec,x)
179         return self.b + torch.sum(self.support_alpha*self.support_label*kernel,dim = 0).view
    (-1,1)#(sample,b)-> b
180 
181     def predict(self, X):
182         """预测标签"""
183         result = torch.sign(self.decision_function(X)) #最终结果按照正负进行二分类，+1，-1
    （根据标签来确定）
184         print(result)
185         return result
186 
187     def score(self, X, y):
188         """计算准确率"""
189         y_pred = self.predict(X).view(-1)        # 展平成 (n,)
190         y_true = y.view(-1)                      # 展平成 (n,)
191         print(f"pred:{y_pred},true:{y_true}")
192         correct = (y_pred == y_true).float()     # 要保证都是向量，要不然可能维度匹配错了
193         acc = correct.mean().item()
194         return acc
195 
196 def predict_k(x, y, modelk):
197 
198     """使用 k 个二分类 SVM 做多分类预测（one-vs-rest 风格）
199     方法:
200     - 对于每个模型，优先使用 model.decision_function(x) 作为得分（连续值，越大越倾向该类），
201       如果没有 decision_function，则使用 model.predict(x) 的 ±1 作为得分回退。
202     - 将所有模型的得分堆叠为形状 (n_samples, k) 的张量 scores。
203     - 对每一行取 argmax 得到预测的类别索引（0..k-1）。
204 
205     返回:
206     - preds: (n_samples,) 的整型张量，表示每个样本被预测为哪个类别（模型索引）
207     - scores: (n_samples, k) 的浮点张量，表示每个样本在每个模型上的得分
208     """
209     y_preds = []
210     for model in modelk:
211         # 优先使用 decision_function 获取连续分数
```

```
212             with torch.no_grad():
213                 s = model.decision_function(x).view(-1).float()  # (n,)
214             y_preds.append(s)
215         #堆叠为 (k, n) -> 转置为 (n, k)
216         scores = torch.stack(y_preds, dim=1)  # (n_samples, k)
217         # argmax 返回每行得分最大的模型索引
218         preds = torch.argmax(scores, dim=1)
219         true_labels = torch.argmax(y, dim=1)
220         # 把张量移动到 CPU 并转换为一维迭代对象，逐元素比较并计数相等项
221         pred_cpu = preds.cpu()
222         true_cpu = true_labels.cpu()
223         equal_count = 0
224         for p, t in zip(pred_cpu, true_cpu):
225             if int(p.item()) == int(t.item()):
226                 equal_count += 1
227         print(f"SVM acc:{equal_count/pred_cpu.size()[0]}")
228         return preds, scores
229 
230     # # 生成简单的二分类数据
231     # def generate_data(n_samples=100,seed1 = 42):
232     #     """生成线性可分的二分类数据"""
233     #     np.random.seed(seed1)
234 
235     #     # 类别1: 均值[2, 2]
236     #     class1 = np.random.randn(n_samples // 2, 2) + np.array([2, 2])
237     #     # 类别2: 均值[-2, -2]
238     #     class2 = np.random.randn(n_samples // 2, 2) + np.array([-2, -2])
239 
240     #     # 合并数据
241     #     X = np.vstack([class1, class2])
242     #     # 标签: 1 和 -1 (SVM标准标签)
243     #     y = np.hstack([np.ones(n_samples // 2), -np.ones(n_samples // 2)])
244 
245     #     return torch.FloatTensor(X), torch.FloatTensor(y).reshape(-1, 1)
246 
247     # x_train,y_train = generate_data(seed1=42)
248     # x_val,y_val = generate_data(seed1 = 100)
249     # net = KernelSVM(in_shape=2)
250     # train_loss,val_loss = net.trainmodel(x_train=x_train,x_val=x_val,y_train=y_train,y_val=
        y_val,epochs=1000)
251     # net.visualize_svm(x_val,y_val,train_loss)
252 
253     def visualize_kernel_svm(model, X, y, losses, title="Kernel SVM", feature_names=None,
        use_pca=True):
254         """可视化核SVM的决策边界和训练过程
255 
256         Args:
257             model: 训练好的KernelSVM模型
```

```
        X: 特征数据 (n_samples, n_features)
        y: 标签 (n_samples, 1) 或 (n_samples,)
        losses: 训练损失列表
        title: 图表标题
        feature_names: 特征名称列表，用于2D可视化时的轴标签
        use_pca: 当特征维度>2时，是否使用PCA降维可视化
    """
    fig = plt.figure(figsize=(16, 5))

    X_np = X.numpy()
    y_np = y.numpy().flatten()

    # 将标签转换为1和-1（如果是0和1）
    if np.any(y_np == 0):
        y_plot = np.where(y_np == 1, 1, -1)
    else:
        y_plot = y_np

    # 处理高维数据：使用PCA降维到2D进行可视化
    if X_np.shape[1] > 2 and use_pca:
        pca = PCA(n_components=2)
        X_2d = pca.fit_transform(X_np)
        explained_var = pca.explained_variance_ratio_
        xlabel = f'PC1 ({explained_var[0]*100:.1f}% var)'
        ylabel = f'PC2 ({explained_var[1]*100:.1f}% var)'
        print(f"使用PCA降维: {X_np.shape[1]}维 -> 2维，保留方差: {sum(explained_var)*100:.2f
    }%")
    elif X_np.shape[1] == 2:
        X_2d = X_np
        if feature_names and len(feature_names) >= 2:
            xlabel, ylabel = feature_names[0], feature_names[1]
        else:
            xlabel, ylabel = 'Feature 1', 'Feature 2'
    else:
        # 如果维度大于2且不使用PCA，则只绘制前两个特征
        X_2d = X_np[:, :2]
        xlabel, ylabel = 'Feature 1', 'Feature 2'
        print(f"警告：仅显示前2个特征，总共{X_np.shape[1]}个特征")

    # 左图：决策边界
    ax1 = plt.subplot(1, 3, 1)

    # 绘制数据点
    ax1.scatter(X_2d[y_plot == 1, 0], X_2d[y_plot == 1, 1],
                c='red', label='good melon(+1)', alpha=0.6, edgecolors='k', s=80)
    ax1.scatter(X_2d[y_plot == -1, 0], X_2d[y_plot == -1, 1],
                c='blue', label='bad melon(-1)', alpha=0.6, edgecolors='k', s=80)

```

```
    # 绘制支持向量（需要投影到2D空间）
    if hasattr(model, 'support_vec') and model.support_vec is not None:
        support_indices = []
        for sv in model.support_vec:
            # 找到原始数据中对应的索引
            for i, x in enumerate(X):
                if torch.allclose(x, sv):
                    support_indices.append(i)
                    break
        if support_indices:
            ax1.scatter(X_2d[support_indices, 0], X_2d[support_indices, 1],
                        s=200, facecolors='none', edgecolors='green', linewidths=2.5,
                        label='Support Vectors')

    # 绘制决策边界（使用原始高维数据进行预测，但在2D空间展示）
    if X_np.shape[1] > 2 and use_pca:
        # 对于PCA降维的情况，在2D空间创建网格，然后反向投影到原始空间
        x1_min, x1_max = X_2d[:, 0].min() - 0.5, X_2d[:, 0].max() + 0.5
        x2_min, x2_max = X_2d[:, 1].min() - 0.5, X_2d[:, 1].max() + 0.5
        xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max, 100),
                               np.linspace(x2_min, x2_max, 100))

        # 反向投影到原始特征空间
        grid_2d = np.c_[xx1.ravel(), xx2.ravel()]
        grid_original = pca.inverse_transform(grid_2d)
        grid_tensor = torch.FloatTensor(grid_original)

        with torch.no_grad():
            Z = model.decision_function(grid_tensor).numpy().reshape(xx1.shape)
    else:
        # 对于2D数据，直接在特征空间绘制
        x1_min, x1_max = X_2d[:, 0].min() - 0.1, X_2d[:, 0].max() + 0.1
        x2_min, x2_max = X_2d[:, 1].min() - 0.1, X_2d[:, 1].max() + 0.1
        xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max, 100),
                               np.linspace(x2_min, x2_max, 100))

        # 如果原始特征多于2个，需要填充其他维度（用均值）
        if X_np.shape[1] > 2:
            grid_2d = np.c_[xx1.ravel(), xx2.ravel()]
            # 添加其他特征的均值
            other_features = np.tile(X_np[:, 2:].mean(axis=0), (grid_2d.shape[0], 1))
            grid_full = np.hstack([grid_2d, other_features])
            grid_tensor = torch.FloatTensor(grid_full)
        else:
            grid_tensor = torch.FloatTensor(np.c_[xx1.ravel(), xx2.ravel()])

        with torch.no_grad():
            Z = model.decision_function(grid_tensor).numpy().reshape(xx1.shape)
```

```
353
354     # 决策边界 (f=0) 和间隔 (f=±1)
355     ax1.contour(xx1, xx2, Z, levels=[0], colors='k', linewidths=2.5, linestyles='-')
356     ax1.contour(xx1, xx2, Z, levels=[-1, 1], colors='gray', linewidths=1.5, linestyles='--',
         alpha=0.7)
357     ax1.contourf(xx1, xx2, Z, levels=[-100, 0, 100], colors=['blue', 'red'], alpha=0.1)
358
359     ax1.set_xlabel(xlabel, fontsize=11)
360     ax1.set_ylabel(ylabel, fontsize=11)
361     ax1.set_title(title, fontsize=12, fontweight='bold')
362     ax1.legend(fontsize=9)
363     ax1.grid(True, alpha=0.3)
364
365     # 中图: 损失曲线
366     ax2 = plt.subplot(1, 2, 2)
367     ax2.plot(losses, linewidth=2, color='purple')
368     ax2.set_xlabel('Epoch', fontsize=11)
369     ax2.set_ylabel('Dual Objective', fontsize=11)
370     ax2.set_title('Training Loss', fontsize=12, fontweight='bold')
371     ax2.grid(True, alpha=0.3)
372
373     # # 右图: 分布
374     # ax3 = plt.subplot(1, 3, 3)
375     # alpha_np = model.alpha.detach().numpy().flatten()
376     # ax3.hist(alpha_np, bins=30, color='orange', alpha=0.7, edgecolor='black')
377     # ax3.axvline(x=0, color='r', linestyle='--', linewidth=1, label=' =0')
378     # ax3.axvline(x=model.C, color='r', linestyle='--', linewidth=1, label=f' =C({model.C})
        ')
379     # ax3.set_xlabel(' value', fontsize=11)
380     # ax3.set_ylabel('Count', fontsize=11)
381     # ax3.set_title('Distribution of Dual Variables ', fontsize=12, fontweight='bold')
382     # ax3.legend(fontsize=9)
383     # ax3.grid(True, alpha=0.3)
384
385     plt.tight_layout()
386     plt.show()
387
388
389 # def generate_linear_data(n_samples=100, seed=42):
390 #     """生成线性可分的数据"""
391 #     np.random.seed(seed)
392 #     class1 = np.random.randn(n_samples // 2, 2) + np.array([2, 2])
393 #     class2 = np.random.randn(n_samples // 2, 2) + np.array([-2, -2])
394 #     X = np.vstack([class1, class2])
395 #     y = np.hstack([np.ones(n_samples // 2), -np.ones(n_samples // 2)])
396 #     return torch.FloatTensor(X), torch.FloatTensor(y)
397
398
```

```python
# def generate_circle_data(n_samples=200, seed=42):
#     """生成同心圆数据（非线性可分）"""
#     np.random.seed(seed)

#     # 内圆（正类）
#     r_inner = np.random.rand(n_samples // 2) * 1.5
#     theta_inner = np.random.rand(n_samples // 2) * 2 * np.pi
#     class1 = np.column_stack([
#         r_inner * np.cos(theta_inner),
#         r_inner * np.sin(theta_inner)
#     ])

#     # 外圆（负类）
#     r_outer = np.random.rand(n_samples // 2) * 1.5 + 2.5
#     theta_outer = np.random.rand(n_samples // 2) * 2 * np.pi
#     class2 = np.column_stack([
#         r_outer * np.cos(theta_outer),
#         r_outer * np.sin(theta_outer)
#     ])

#     X = np.vstack([class1, class2])
#     y = np.hstack([np.ones(n_samples // 2), -np.ones(n_samples // 2)])
#     return torch.FloatTensor(X), torch.FloatTensor(y)


# def generate_xor_data(n_samples=200, seed=42):
#     """生成XOR数据（非线性可分）"""
#     np.random.seed(seed)
#     X = np.random.randn(n_samples, 2) * 2
#     y = np.ones(n_samples)
#     y[(X[:, 0] * X[:, 1]) < 0] = -1  # XOR pattern
#     return torch.FloatTensor(X), torch.FloatTensor(y)


if __name__ == '__main__':#西瓜数据集上的测试
    data = pd.read_csv('xigua3_0.csv')
    #data.drop(columns=['编号'],inplace=True)#inplace:直接在原数据上修改
    data['好瓜'].replace(['是', '否'], [1, -1], inplace=True)  # SVM需要1和-1标签
    all_features = pd.get_dummies(data, dummy_na=True)#dummy_na会把缺失值也单开一列，作为独立的特征（哑变量处理）
    all_features.drop(columns=['好瓜'],inplace=True)
    all_features = all_features.astype(float)
    train_features = torch.tensor(all_features.values, dtype=torch.float32)
    train_labels = torch.tensor(data['好瓜'].values, dtype=torch.float32).view(-1, 1)#一维转二维，本来就是二维就不用转换了

    model_poly = KernelSVM(input_dim=train_features.shape[0], C=1,sigma=1)  # 设置sigma:使用高斯核,设置d:使用多项式核，不设置:线性核
```

```
444     losses_poly = model_poly.trainmodel(train_features, train_labels, epochs=1000)
445     accuracy_poly = model_poly.score(train_features, train_labels)
446     print(f"Training Accuracy: {accuracy_poly * 100:.2f}%\n")
447 
448     visualize_kernel_svm(model_poly, train_features, train_labels, losses_poly,
449                          title=f"SVM-watermelon(acc:{accuracy_poly*100:.1f}%)") #PCA因为投影到了2维平面空间，并不一定准确表征分类的结果，acc现实的才是真正的准确率，实验来看，二维多项式核的结果最好，RBF次之，最不好的是线性核
```