数据的折叠与归并：均值聚类模式观点下的

主成分分析、KNN 分类和 SVM 核方法

20 级软件工程辅修班 F201930075021 刘华林

数据折叠与归并定义：

在变量比较多时，样本就是高维空间中的点，只要假设样本的分布是具有模式的（也即高维空间中的点分布在一些几何体上，满足这些几何体形状的概率分布），那么通过对这个空间施加一定的线性或非线性变换，就可以扭曲这些几何体的形状，将这些几何体（样本的真实分布）的表达式变得简单；或者将一些变量合并归总进行降维，也可以将这些几何体（样本的真实分布）的表达式变得简单，这就是**数据特征的折叠**。

设样本空间正交基为$(x_1, x_2, x_3, \cdots, x_n)$，构造一个折叠映射$\Phi: (x_1, x_2, x_3, \cdots, x_n) \rightarrow (x_p, \cdots, x_q)$，使得$|p-q| < n$，$\lim\limits_{m\to\infty} |y - \hat{y}| = 0$，$m$为样本数目。**折叠映射Φ具有保一致性：分类器/回归器的估计概率分布在变换前后的样本空间中随着样本数量增大都可以收敛到真实分布**。当然，保一致性是最基本的要求，更好的折叠映射Φ应具备更好的统计性质。

从数据生成的观点，不仅变量（特征、维度）可以折叠，样本数据点也是可以折叠的，假设样本的确可以被某些工具良好地分类，那么在这些分类方法下，样本实际上可以表达为更少乃至很少的数据点加以一定的随机性。例如如果一组数据能用线性回归极高精度地拟合，那么这组样本实际上也可以为线性回归+高斯噪声很好地生成，假设可以良好地分类/回归的前提下用分类器/回归器+噪声表示数据，这是**数据的归并**。

设样本集合为$(x_1, x_2, x_3, \cdots, x_n)$，构造分类器/回归器$F$，使得$\forall x_i, |F(x_i) - y_i| \sim N(0,1)$，选择高斯分布$N(0,1)$作为噪声是因为高斯分布噪声是在测量过程中自然存在、不可避免的，无法通过构造新的分类器/回归器$F' = F + N(0,1)$来回避，对于非加性$N(0,1)$的噪声（非高斯噪声的，或者异均值异方差的）则归纳到项F中。

构造一阶生成器$G^1 = F(x_i) + N(\mu_{G^1}, {\sigma^2}_{G^1}), x_i \in (x_1, x_2, x_3, \cdots, x_n)$，得到新的样本集合$\left(x_{G_1^1}, x_{G_2^1}, x_{G_3^1}, \cdots, x_{G_N^1}\right)$，这些新样本与原样本只相差一个已知的高斯噪声，可用$F = F' - N(0,1) = G^1 - N(\mu_{G^1}, {\sigma^2}_{G^1})$进行估计。同样可以构造二阶、三阶、……N 阶生成器，一阶生成器对原数据的误差$e_1 \sim N(0,1)$，二阶生成器对一阶生成器误差$e_2 \sim N(0,1)$，到第 N 阶生成器，累计误差$\sum_{i=1}^{N} e_i \sim \chi^2(N)$，$E[\chi^2(N)] = N$，累计误差期望会随生成器阶数提升的过程而线性扩大。

假设原始数据$(x_1, x_2, x_3, \cdots, x_n) = \hat{X}$是可以表示总体的真实情况的，只是$\hat{X}$包含了抽样过程的噪声，$\hat{X}$作为总体的一个抽样，假定$\hat{X}$上的估计分布$F$离真实分布$\tilde{F}$只差一个未知的正态分布$N(\mu, \sigma^2)$，那么不断从$F$上构造生成器$G^i$，由于累计误差（偏离）$\sum_{i=1}^{N} e_i$的期望线性扩大，**最终在生成器阶数和未知正态分布的均值不断靠近时，G^i可以获得比$\hat{X}$更真实的总体抽样**，但在这么做之前必须平衡生成器数值的数量级，进行数据标准化。

均值聚类的启发：

K 均值聚类是从局部出发随机选取数据点作为初始聚类中心，再依次访问各个数据点计算与各聚类中心数据点的距离，将分类到最小距离的中心所属的类；再针对分类好的类别重新计算这个类别的所有数据点的聚类中心（质心），再依次访问各数据点进行归类；不断如此做直到聚类中心们的位置变化很小。

均值聚类是一个很好的玩具模型（Toy Model），**聚类本身是一种构造一个分类器的操作，基于分类器可以对样本数据集进行归并**；同时聚类过程要求根据样本数据集构造样本空间，样本数据点具有多个维度：这些维度可能数量级有差异，所遵循的真实分布不同，因此如果直接按原数据集生成样本空间，样本的数据点可能会杂乱地分布，这时可以**引入一定的先验经验，重新构造样本空间的正交基、线性/非线性地拉伸和扭曲空间形状、升维到特殊的高维空间**等，得到样本数据点分布更加有序的新样本空间，使聚类过程变得快而简单。

从这种思想出发，我想展示三个经典的机器学习方法：主成分分析，KNN 最近邻算法和 SVM 支持向量机方法。这三类方法都存在着均值聚类的思想：

1. 主成分分析基于方差将维度“聚类”为主成分，最大方差的主成分是主成分类的中心，根据数据归并的思想，只需要构造生成器（只需更少部分的数据）就可以模拟真实分布，也即构造少数几个大方差的主成分，就可以提取主要的信息；
2. KNN 最近邻算法，通过最优化方法找到一个合适的 K，可以让算法捕捉样本空间内的合适的“形状”，K 的大小决定了这种捕捉的精细程度与误判概率：很小的 K 非常精细但容易“过拟合”在局部误判，很大的 K 则会“欠拟合”产生更多的全局误判。K 就如同均值聚类的聚类数目决定着拟合程度；
3. 支持向量机一般仅用于二分类问题，SVM 仅需超平面边界附近的几个数据点即可决定整个数据集的“形状”，就如同均值聚类的聚类中心一样，可以用中心点+噪声来生成样本数据点。此外，SVM 的核方法也是一种变换特征空间的经典手段，将非线性边界问题转化为高维空间的线性可分边界问题，类似于均值聚类可以变换特征空间和距离度量，加速聚类计算过程。

主成分分析的数据折叠方法：

主成分分析是一种特征降维手段，首先主成分分析需要标准化的数据，进行数据预处理，可以对该变量下的样本求均值和方差得到标准化值：

$$x' = \frac{x - \mu}{\sigma}$$

主成分分析有“散度最大化”的假设，总是优先选择数据点方差最大的变量作为次序更靠前的主成分，这样就可以达成更高的区分能力，并且这些主成分是彼此正交的（就像正交的各变量维度一样），这样就可以用更少的维度（主成分维度）构成一个新的样本空间，只要假设方差在各变量维度的分布的确是不均匀的，那么新的主成分空间确实会比原空间更小。

在计算中，对于样本的数据矩阵，为求得各变量下的方差，可以求数据矩阵的协方差矩阵得到数据的方差分布，协方差矩阵的对角阵（仅在对角线上元素非零）上特征值最大的维度就是方差最大的维度，由于一般样本数与变量数并不一致，传统的特征矩阵分解只适用于方阵，求该对角阵可以采用 SVD 矩阵分解的方法，以下是 numpy 下的实现代码：

```
import numpy as np
#X 为数据矩阵，k 为主成分个数
def pca(X,k):
    #X.shape = (m,n),m 为行数样本数,n 为列数变量数
    n_samples, n_features = X.shape
    #对 X 每列变量求均值得到均值向量
    mean = np.array([np.mean(X[:,i]) for i in range(n_features)])
    #标准化
    norm_X = X - mean
    #计算 X·X^T 得到协方差矩阵
```

```
    scatter_matrix = np.dot(np.transpose(norm_X),norm_X)
    #计算方阵的特征值
    eigen_val, eigen_vec = np.linalg.eig(scatter_matrix)
    #整理成对
    eigen_pairs = [(np.abs(eigen_val[i]),eigen_vec[:,i]) for i in range(n_features)]
    #按特征值（方差）从大到小排列
    eigen_pairs.sort(reverse=True)
    #选择最大的 k 个特征
    feature = np.array([element[1] for element in eigen_pairs[:k]])
    #得到新的数据矩阵
    new_X = np.dot(norm_X,np.transpose(feature))
    return new_X
X = np.array([[-1,1],[-2,-1],[-3,-2],[1,1],[2,1],[3,2]])
print(X)
print(pca(X,1))
```

运行结果：$x=\begin{bmatrix}-1 & 1\\ -2 & -1\\ -3 & -2\\ 1 & 1\\ 2 & 1\\ 3 & 2\end{bmatrix}$ $y=\begin{bmatrix}-0.50917706\\ -2.40151069\\ -3.7751606\\ 1.20075534\\ 2.05572155\\ 3.42937146\end{bmatrix}$

可以看到，x 的两个变量维度被整合为一个主成分，压缩到了高方差的主成分维度。

KNN 最近邻算法的数据归并方法：

主成分分析是从比较不同维度下样本的方差出发进行“数据折叠”的，是全局性的，而 KNN(K-Nearest Neighbour)算法则是启发式地从局部（依次访问每个数据点计算距离）来确定数据范围。KNN 方法有三大要素：

1. K 值选择，K 表示计算分类时纳入参考的周围数据点的个数，K 值小容易产生扭曲的局部曲线和形状：过拟合，K 值大容易形成平缓的局部曲线和形状：欠拟合，需要确定一个好的 K 值；
2. 距离度量方法，在 N 个变量维度下，当 N 较大时，KNN 算法发生在高维空间，各数据点的分布会比较稀疏、欧氏距离效果不好，可以采用其他距离度量或者用主成分等方法降维缓解维数爆炸问题；
3. 分类规则，根据周围数据点的类型对当前数据点进行分类，一般采用多数表决的原则，当前数据点被划为类别最多的数据点的类。

显然 KNN 算法能很好地可视化数据点的分类，但可以用距离加权方法（K 个最近邻居结合他们的距离度量值求和）预测连续值。结合 KNN 的可视化分类功能和预测性，可以使用 KNN 对异常值进行检测，异常数据点在可视化图形中是离群的孤岛或者单点，异常值出现的原因可能是信息来源的伪造或者 KNN 泛化能力不够。

以下是 PyTorch 的实现代码：

```
from torchvision import datasets, transforms
import numpy as np
from sklearn.metrics import accuracy_score
import torch
import time
```

```
#KNN 算法，参数为训练与测试数据集，图像为 Tensor，标签为 List，初始 k 值
def KNN(train_x,train_y,test_x,test_y,k):
    #开始计时
    since = time.time()
    #第一个张量维度的数值
    m = test_x.size(0)
    n = train_x.size(0)
    #求 Euclid 距离矩阵
    print(">>计算欧氏距离矩阵")
    #测试数据求平方，并按列求和
    x2 = (test_x**2).sum(dim=1,keepdim=True).expand(m,n)
    y2 = (train_x**2).sum(dim=1,keepdim=True).expand(n,m).transpose(0,1)
    #求距离矩阵：x^2 + y^2 - 2xy
    dist_mat = x2 + y2 - 2*test_x.matmul(train_x.transpose(0,1))
    #在最后面的维度进行排序，返回下标矩阵
    min_k_idxs = dist_mat.argsort(dim=-1)
    res = []
    for idxs in min_k_idxs:
        #分类规则
        res.append(np.bincount(np.array([train_y[idx] for idx in idxs[:k]])).argmax())
    print("-"*8,"结果评估","-"*8)
    print("精准度：",accuracy_score(test_y,res))
    time_elapsed = time.time() - since
    print("KNN 训练完成：{:.3f}ms".format(time_elapsed))

#主过程
if __name__ == "__main__":
    #采用 MNIST 手写字体数据集，若未下载可设置参数 download=True，下载到我的文档/data 处
    train_dataset =
datasets.MNIST(root="./data",transform=transforms.ToTensor(),train=True,download=True)
    test_dataset =
datasets.MNIST(root="./data",transform=transforms.ToTensor(),train=False,download=True)
    #训练数据部分：图片 x 和标签 y
    train_x = []
    train_y = []
    for i in range(len(train_dataset)):
        img, target = train_dataset[i]
        train_x.append(img.view(-1))
        train_y.append(target)
        #收集 5000 个图片即可
        if i>5000:
            print(">>训练集的 5000 个样本载入")
            break
    test_x = []
```

```
test_y = []
for i in range(len(test_dataset)):
    img, target = test_dataset[i]
    test_x.append(img.view(-1))
    test_y.append(target)
    #对 200 组数据测试即可
    if i>200:
        print(">>测试集的 200 个样本载入")
        break
#分 7 个类别
KNN(torch.stack(train_x),train_y,torch.stack(test_x),test_y,8)
```

运行结果：

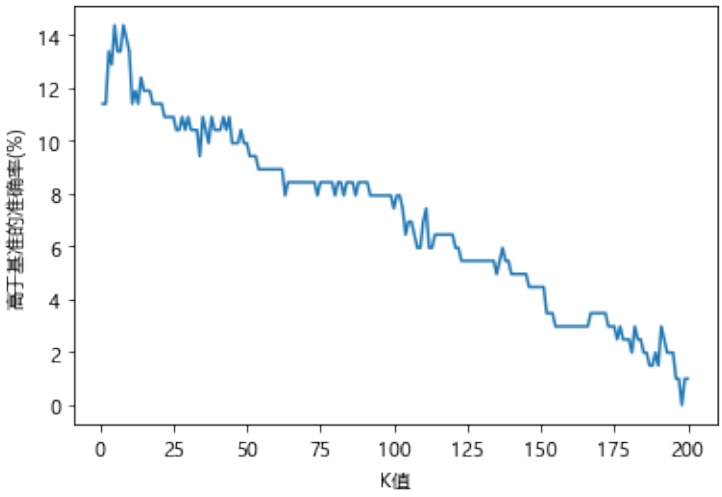
```
>>训练集的 5000 个样本载入
>>测试集的 200 个样本载入
>>计算欧氏距离矩阵
-------- 结果评估 --------
精准度： 0.9504950495049505
KNN 训练完成：0.105ms
```

用 MNIST 手写字体图片数据集（5000 个二维图片样本用于验证），计算 8 个最近邻居得到 202 个图片特征空间中的类别，>0.95 的分类准确率，现在分别对不同 K 值（K∈[1,10]）的分类精准度进行计算：

K 值	分类精准度
1	0.9207920792079208（92.08%）
2	0.9207920792079208（92.08%）
3	0.9405940594059405（94.06%）
4	0.9356435643564357（93.56%）
5	0.9504950495049505（95.05%）
6	0.9405940594059405（94.06%）
7	0.9405940594059405（94.06%）
8	0.9504950495049505（95.05%）
9	0.9455445544554455（94.55%）
10	0.9405940594059405（94.06%）

对 1~200 的 K 值情形进行计算可以得到：

```
最低准确度： 0.806930693069307（80.69%）
最高准确度： 0.9504950495049505（95.05%）
```



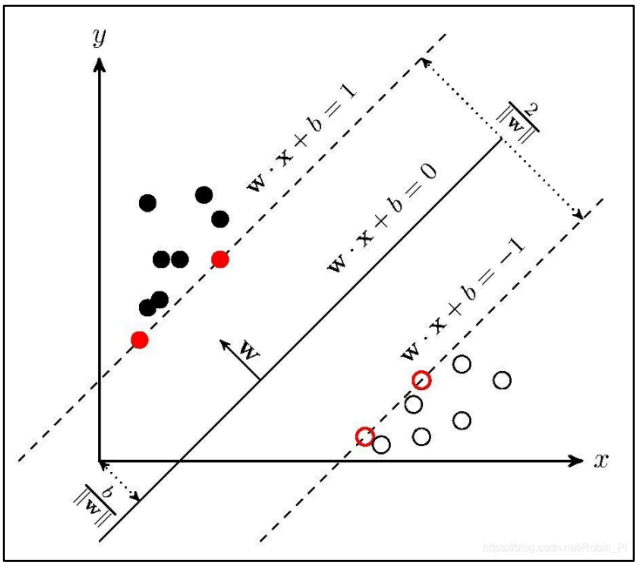
上图横轴为 K 值，纵轴为 K 值对应的分类准确度、相比最低准确度情况高多少个百分点。可以观察到，随着 KNN 判定的最近数据点个数越来越多，分类准确度呈台阶式下降，K=5 与 K=8 是最高准确度，其余的 K<5 为过拟合情形，K>8 为欠拟合情形。

SVM 支持向量机的数据归并与折叠方法：

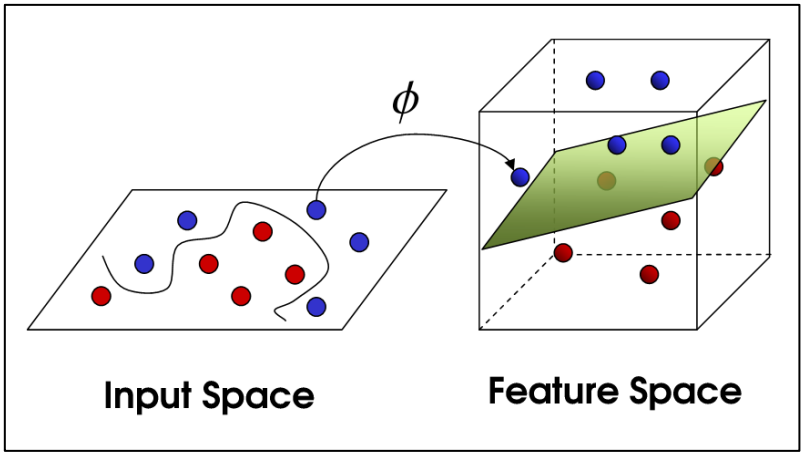
KNN 分类是一种启发式的分类方法，所有数据点参与类的形成，而 SVM（支持向量机）则只有部分数据点参与类的形成（通过确定两个类的边界），仅用超平面$\vec{w}^T\vec{x}+b=0$对整个平面做划分，在整体上将所有数据点划分到两个标签。样本空间中（在这里是两个特征的二维样本空间）任意数据点到超平面距离：

$$r_i=\frac{|\vec{w}^T\vec{x_i}+b|}{||\vec{w}||}$$

SVM 的目标是最大化r_i，也即最小化$||\vec{w}||$，可知目标函数为$arg_{min}\left(\frac{1}{2}||\vec{w}||^2\right)$，这里$||\cdot||^2$表示向量的 2-范数。



训练完毕的 SVM 分类器应保证$|\vec{w}^T\vec{x_i}+b|>1$，才可保证没有数据点在两个超平面中间。然而常见的问题是数据点并不像上图一般有良好的线性可分性，如下图所示：



二维样本空间的数据点分布的并不规整，但也可能存在一个区分边界的曲线。数学上有一个 Cover 定理：将复杂的模式分类问题非线性地投射到高维空间将比投射到低维空间更可能是线性可分的，当空间的维数 D 越大时，在该空间的 N 个数据点间的线性可分的概率就越大。类比上图，用映射Φ将二维样本空间变换到三维，样本之间彼此的距离加大了，并显示出更平直（线性可分）的数据区分边界。

对二维样本空间的特征x, y，映射到三维样本空间后分别为$\Phi(x), \Phi(y), z$，其中z是一个未知的新特征维度。令$\kappa(x, y) = \langle \Phi(x), \Phi(y) \rangle$，其中$\langle \cdot \rangle$为计算内积，$\kappa(x, y)$为核函数，核函数直接获取特征映射后计算的内积结果，而无需理会高维空间中的计算细节。

很明显，既然核函数$\kappa(x, y)$可以叠加特征映射$\Phi(\cdot)$的计算和内积$\langle \cdot \rangle$的计算，核函数与特征映射具有对应的关系，我们如何映射到高维空间的方式决定了使用什么核函数。在这个问题上另有 Mercer 定理：任何半正定对称函数都可以作为核函数，函数的正定性可以保持 SVM 原有的凸优化性质，维持在新的高维特征空间中的凸优化性质。

以下是 SVM 的 Pytorch 实现代码：

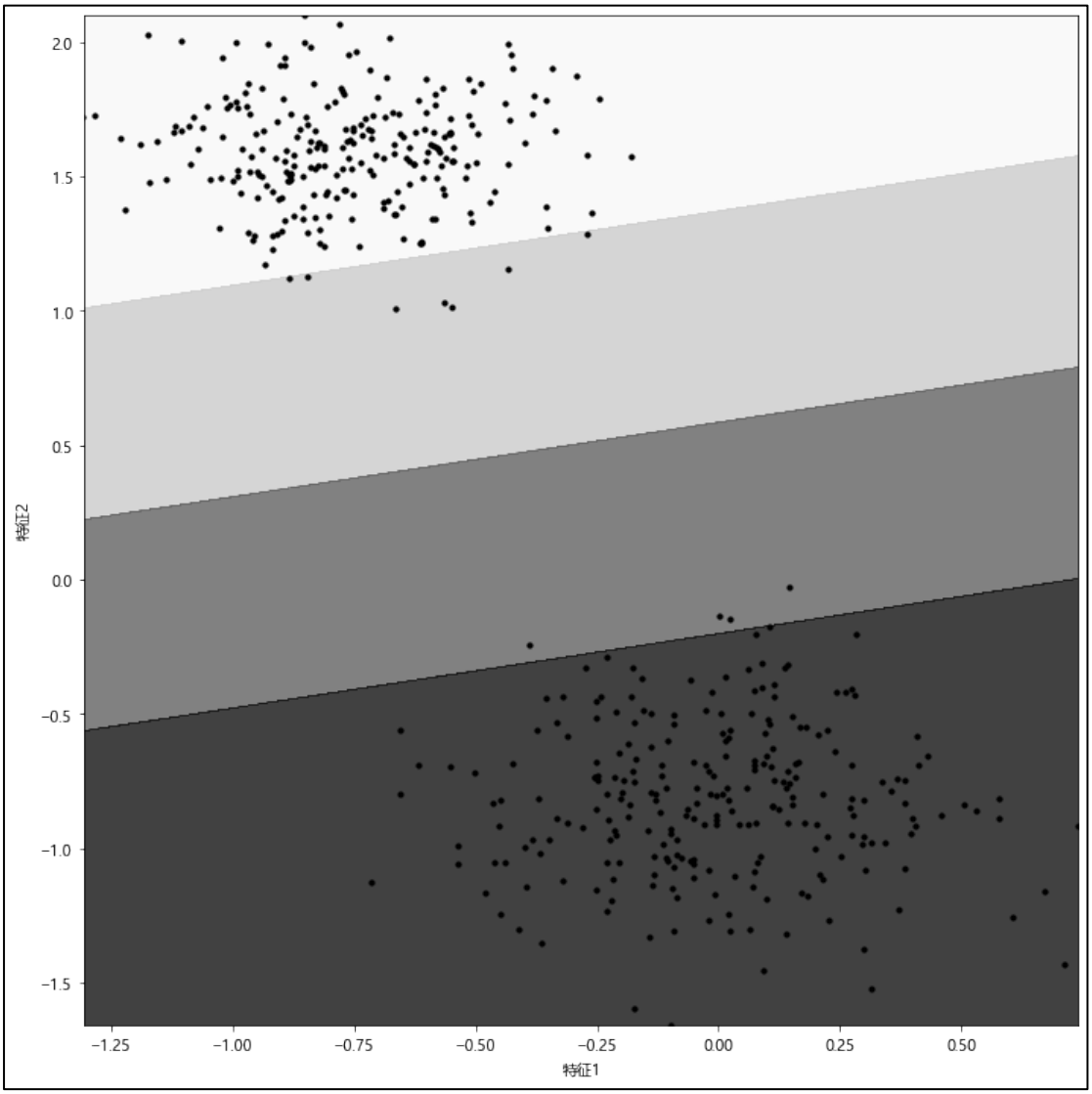
```
import argparse
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets._samples_generator import make_blobs

def train(X,Y,model,args):
    X = torch.FloatTensor(X)
    Y = torch.FloatTensor(Y)
    N = len(Y)
    #创建随机梯度下降优化器
    optimizer = optim.SGD(model.parameters(),lr=args.lr)
    #使用 model.train()可激活批归一化和 Dropout 技术
    model.train()
    #迭代每一轮
    for epoch in range(args.epoch):
        #得到 0~N-1 的随机整数列表
        perm = torch.randperm(N)
```

```
            sum_loss = 0.0
            #按批访问数据
            for i in range(0,N,args.batchsize):
                x = X[perm[i:i+args.batchsize]].to(args.device)
                y = Y[perm[i:i+args.batchsize]].to(args.device)
                #优化前梯度归零
                optimizer.zero_grad()
                output = model(x).squeeze()
                weight = model.weight.squeeze()
                #计算损失
                loss = torch.mean(torch.clamp(1-y*output,min=0))
                loss += args.c * (weight.t() @ weight) / 2.0
                loss.backward()
                optimizer.step()
                sum_loss += float(loss)
            print("轮数：{:4d}\t 损失：{:.6f}".format(epoch,sum_loss/N))
        return
    def visualize(X,Y,model):
        W = model.weight.squeeze().detach().cpu().numpy()
        b = model.bias.squeeze().detach().cpu().numpy()
        delta = 0.001
        #根据 X，Y 的范围生成网格数据
        x = np.arange(X[:,0].min(),X[:,0].max(),delta)
        y = np.arange(X[:,1].min(),X[:,1].max(),delta)
        x, y = np.meshgrid(x,y)
        x_y = list(map(np.ravel,[x,y]))
        #计算等高图（热力图）
        z = (W.dot(x_y)+b).reshape(x.shape)
        z[np.where(z>1.0)] = 4
        z[np.where((z>0.0) & (z<=1.0))] = 3
        z[np.where((z>-1.0) & (z<=0.0))] = 2
        z[np.where(z<=-1.0)] = 1
        plt.figure(figsize=(10,10))
        plt.xlim([X[:,0].min()+delta,X[:,0].max() - delta])
        plt.ylim([X[:,1].min()+delta,X[:,1].max() - delta])
        plt.contourf(x,y,z,alpha=0.8,cmap="Greys")
        #绘制训练样本的散点
        plt.scatter(x=X[:,0],y=X[:,1],c="black",s=10)
        plt.tight_layout()
        plt.show()
        return
    if __name__ == "__main__":
        #生成命令行参数对象
        parser = argparse.ArgumentParser()
```

```
    parser.add_argument("--c",type=float,default=0.01)
    parser.add_argument("--lr",type=float,default=0.1)
    parser.add_argument("--batchsize",type=int,default=5)
    parser.add_argument("--epoch",type=int,default=10)
    parser.add_argument("--device",default="cpu",choices=["cpu","cuda"])
    args = parser.parse_known_args()[0]
    args.device = torch.device(args.device if torch.cuda.is_available() else "cpu")
    #make_blobs 生成数据集，n_samples 为样本个数，centers 为标签数，cluster_std 为某个标签下样本的方差，
random_state 为随机种子
    X, Y = make_blobs(n_samples=500,centers=2,random_state=0,cluster_std=[0.3,0.4])
    #X 矩阵(500 行 2 列)数值进行标准化
    X = (X - X.mean()) / X.std()
    #(0,1)标签改为(-1,1)标签
    Y[np.where(Y==0)] = -1
    #生成线性层神经网络，输入 2 个特征，输出 1 个特征
    model = nn.Linear(2,1)
    #模型加载到设备上
    model.to(args.device)
    #训练模型
    train(X, Y, model, args)
    #可视化输出
    visualize(X,Y,model)
```

运行结果:

轮数	损失
1	0.006610
2	0.002434
3	0.002354
4	0.002313
5	0.002322
6	0.002302
7	0.002337
8	0.002287
9	0.002302
10	0.002306



最终训练 10 轮得到的 SVM 分类器的超平面方程：

$$[0.3524, -1.2725] \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + 0.7430 = 0$$

代码采用 make_blobs 生成一个带标签的有方差的数据集，并将标签从[0,1]变为[-1,1]将数据点拉出分界线。在 2 个特征的 2 分类数据集上运行上述的 SVM，分类效果极好。

结束