基于软间隔 SVM 算法对不同品种种子分类

1. 问题描述

在 UCI 的数据集 seeds Data Set 中，包含了三种不同小麦品种的籽粒：卡马、罗莎和加拿大，每种 70 种元素，随机选择用于实验，由于实验要求，我只选取前两种籽粒（卡马，罗莎）用于分类。本文将运用软间隔 SVM 算法，利用 seeds Data Set 中作者通过使用软 X 射线技术检测到内核内部结构所得到的这 7 维特征数据来建立一个分类模型，可以帮助我们分辨出测试样本是卡马籽粒还是罗莎籽粒。当然若将卡马与加拿大籽粒混合、罗莎与加拿大籽粒混合同样也能将其鉴别。

数据集地址：https://archive.ics.uci.edu/ml/datasets/seeds

2. 算法原理

由于 seeds Data Set 的种子样本中有三种类型的种子，而 SVM 算法用于二分类问题较为普遍，因此我只选择了两种类型的种子进行分析。为了方便模型训练，我们需要在训练过程中告知机器我们输入的样本的类别，因此本文将卡马类型的种子划分为“1”类，罗莎类型的种子划分为“-1”类。在训练过程中，我们需要向模型输入 X 和 Y，其中 X 为 m*n 的矩阵(m 为样本数量，n 为特征个数)，Y 为 m*1 的矩阵，SVM 中同样包含两个训练参数 W 和 b，其中 W 为 n*1 的矩阵

我们知道，对于硬间隔 SVM 算法，它可以在严格线性可分的数据集上工作的很好，但对于非严格线性可分的情况往往表现不尽如人意。哪怕仅含有一个异常点，对硬间隔支持向量机的训练影响就很大，我们希望它能具有一定的包容能力，能够容忍那些放错的点，但又不能容忍过度，因此我们对硬间隔 SVM 进行改进，从而形成软间隔 SVM 算法。软间隔 SVM 算法引入了变量ξ和一个超参 C 来进行控制，将原始的优化问题更新为如下:

$$\begin{aligned} &\min_{w,b,\xi} \frac{1}{2} w^T w + C \sum_{i=1}^{N} \xi_i \\ &s.t.\ \ y_i(w^T x_i + b) \geq 1 - \xi_i \\ &\xi \geq 0 \end{aligned}$$

:

- 这里的ξ被称为松弛变量，即适当放松原来问题的要求，且每个x_i都有一个对应的$\xi\ i$
- 这里的 C 被称为惩罚参数，且 C > 0。C 越大，表示对误分类点的惩罚越大，迫使所有样本均满足约束，即尽可能将其两组点分开，反之允许更多的点跨过分界线。

根据前面硬间隔的方式，依然将其转化为对偶形式进行求解。首先引入拉格朗日乘子:

$$L(w,b,\xi,\alpha,\beta) = \frac{1}{2} w^T w + C \sum_{i=1}^{N} \xi_i + \sum_{i=1}^{N} \alpha_i (1 - \xi_i - y_i(w^T x_i + b)) + \sum_{i=1}^{N} \beta(-\xi)$$

由 KKT 条件得，原问题的对偶问题为:

$$\max_{\alpha,\beta} \min_{w,b,\xi} L(w,b,\xi,\alpha,\beta)$$
$$s.t.\ \alpha_i \geq 0, i = 1,2,...,N$$
$$\beta_i \geq 0, i = 1,2,...,N$$

将L函数分别对w,b,ξ求导并令其为 0，则有：

$$\begin{cases} w = \sum_{i=1}^{N} \alpha_i y_i x_i \\ \sum_{i=1}^{N} \alpha_i y_i = 0 \\ \beta_i = C - \alpha_i \end{cases}$$

带入$L(w,b,\xi,\alpha,\beta)$，得：

$$L(w,b,\xi,\alpha,\beta) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j x_i x_j + \sum_{i=1}^{N}\alpha_i$$

故，原问题可转化为：

$$\min_{\alpha} \ \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j x_i x_j - \sum_{i=1}^{N}\alpha_i$$
$$s.t. \ \sum_{i=1}^{N}\alpha_i y_i = 0$$
$$0 \leq \alpha_i \leq C$$

由 KKT 条件中的关系 1，我们可以知道：

$$w^* = \sum_{i=1}^{N} \alpha_i^* y_i x_i$$

对于 b*的求解，我们可以取某点，让值为 0：

$$b^* = y_k - w^{*T} x_k$$

后面就是和硬间隔 SVM 一样的使用 SMO 算法求得 α，再求出 w 和 b，与硬间隔 SVM 的对偶形式相比，仅多了 0 ≤ α_i ≤ C 这一个限制条件，而这个参数一般由人为设定，即超参数。

3. 代码展示

数据预处理

首先我先对数据进行预处理，将卡马类型的种子划分为“1”类，罗莎类型的种子划分为“-1”类，由于其样本数量都为 70，因此不需要进行其他处理：

12.78	13.57	0.8716	5.262	3.026	1.176	4.782	1
12.88	13.5	0.8879	5.139	3.119	2.352	4.607	1
14.34	14.37	0.8726	5.63	3.19	1.313	5.15	1
14.01	14.29	0.8625	5.609	3.158	2.217	5.132	1
14.37	14.39	0.8726	5.569	3.153	1.464	5.3	1
12.73	13.75	0.8458	5.412	2.882	3.533	5.067	1
17.63	15.98	0.8673	6.191	3.561	4.076	6.06	-1
16.84	15.67	0.8623	5.998	3.484	4.675	5.877	-1
17.26	15.73	0.8763	5.978	3.594	4.539	5.791	-1
19.11	16.26	0.9081	6.154	3.93	2.936	6.079	-1
16.82	15.51	0.8786	6.017	3.486	4.004	5.841	-1
16.77	15.62	0.8638	5.927	3.438	4.92	5.795	-1
17.32	15.91	0.8599	6.064	3.403	3.824	5.922	-1
20.71	17.23	0.8763	6.579	3.814	4.451	6.451	-1
18.94	16.49	0.875	6.445	3.639	5.064	6.362	-1
17.12	15.55	0.8892	5.85	3.566	2.858	5.746	1

之后为了后续实验的方便和结果有效，将这些数据随机打乱，并按照 4：1 设置训练集与测试集。

```
data = pd.read_csv('seeds_dataset.csv', header=None, sep=',' )
data = np.array(data)
np.random.seed(3)
#随机打乱样本
np.random.shuffle(data)
data_train = data[:110, :] #选取训练样本
data_test = data[110:, :]#选取测试样本
#print(data_test)
X = data_train[:, :7]
Y = data_train[:, 7:]
#print(Y)
X_test = data_test[:, :7]
Y_test = data_test[:, 7:]
```

初始化代码

数据集划分好后，我们需要构建软间隔 SVM，首先需要初始化 SVM 所包含的所有参数(包括迭代次数和超参数 C 等等)，初始化代码如下图所示：

```
#初始化各项参数
def __init__(self, epochs=100, C=1.0):
    self.w = None
    self.b = None
    self.alpha = None
    self.E = None
    self.epochs = epochs
    self.C = C
    # 记录支持向量
    self.support_vectors = None

def init_params(self, X, y):
    n_samples, n_features = X.shape    # 记录样本数和特征数
    self.w = np.zeros(n_features)      # 初始化权值矩阵
    self.b = 0
    self.alpha = np.zeros(n_samples)   # 初始化ai
    self.E = np.zeros(n_samples)       # 初始化误差
    for i in range(0, n_samples):
        self.E[i] = np.dot(self.w, X[i, :]) + self.b - y[i]
```

确定α_i选择α_j

在α_i已经固定好的情况下，对于α_j，我们倾向于选择使其变化尽可能大的点所对应的α，即优先选择使得$|Ei - Ej|$最大的 j。

```python
# SMO中固定aj的值
def select_j(self, best_i):
    # 设定可以取到的j的合理范围
    valid_j_list = [i for i in range(0, len(self.alpha)) if self.alpha[i] > 0 and i != best_i]
    best_j = -1
    # 优先选择使得|Ei-Ej|最大的j，即变化最大的点
    if len(valid_j_list) > 0:
        max_e = 0
        for j in valid_j_list:
            current_e = np.abs(self.E[best_i] - self.E[j])
            if current_e > max_e:
                best_j = j
                max_e = current_e
    # 否则随机选择
    else:
        list1 = list(range(len(self.alpha)))
        seq = list1[: best_i] + list1[best_i + 1:]
        best_j = random.choice(seq)
    return best_j
```

KKT 条件检验

为了固定 α1，我们需要找到不满足 KKT 条件的点，因此，我们需要设置一个判断函数，可以帮助我们判定样本各点是否都满足 KKT 条件。

```python
# 判定各点是否满足kkt条件
def meet_kkt(self, w, b, x_i, y_i, alpha_i):
    if alpha_i < self.C:
        return y_i * (np.dot(w, x_i) + b) >= 1
    else:
        return y_i * (np.dot(w, x_i) + b) <= 1
```

更新 w，b 和α

```python
def fit(self, X, y2):
    y = copy.deepcopy(y2)
    y[y == 0] = -1
    self.init_params(X, y)
    for _ in range(0, self.epochs):        # 不断迭代直到所有点满足kkt条件
        if_all_match_kkt = True
        # 搜寻违反KKT条件的点i，并更新它的alpha值
        for i in range(0, len(self.alpha)):
            x_i = X[i, :]
            y_i = y[i]
            alpha_i_old = self.alpha[i]
            E_i_old = self.E[i]
            if not self.meet_kkt(self.w, self.b, x_i, y_i, alpha_i_old):
                if_all_match_kkt = False
                best_j = self.select_j(i)
                alpha_j_old = self.alpha[best_j]
                x_j = X[best_j, :]
                y_j = y[best_j]
                E_j_old = self.E[best_j]
            # 对选好的两个alpha进行更新
            # 首先获取无裁剪的最优alpha_j
            eta = np.dot(x_i - x_j, x_i - x_j)
            # 如果x_i和x_j很接近，则跳过
            if eta < 1e-3:
                continue
            alpha_j_unc = alpha_j_old + y_j * (E_i_old - E_j_old) / eta
```

```
# 裁剪并得到new alpha_j
if y_i == y_j:
    L = max(0., alpha_i_old + alpha_j_old - self.C)
    H = min(self.C, alpha_i_old + alpha_j_old)
else:
    L = max(0, alpha_j_old - alpha_i_old)
    H = min(self.C, self.C + alpha_j_old - alpha_i_old)
if alpha_j_unc < L:
    alpha_j_new = L
elif alpha_j_unc > H:
    alpha_j_new = H
else:
    alpha_j_new = alpha_j_unc
# 如果变化不够大则跳过
if np.abs(alpha_j_new - alpha_j_old) < 1e-5:
    continue
# 得到alpha_i_new
alpha_i_new = alpha_i_old + y_i * y_j * (alpha_j_old - alpha_j_new)
# 更新w
self.w = self.w + (alpha_i_new - alpha_i_old) * y_i * x_i + (alpha_j_new - alpha_j_old) * y_j * x_j
# 更新alpha_i,alpha_j
self.alpha[i] = alpha_i_new
self.alpha[best_j] = alpha_j_new
# 更新b
b_i_new = y_i - np.dot(self.w, x_i)
b_j_new = y_j - np.dot(self.w, x_j)
if self.C > alpha_i_new > 0:
    self.b = b_i_new
elif self.C > alpha_j_new > 0:
    self.b = b_j_new
else:
    self.b = (b_i_new + b_j_new) / 2.0
# 更新E
for k in range(0, len(self.E)):
    self.E[k] = np.dot(self.w, X[k, :]) + self.b - y[k]
```

```
        # 如果所有的点都满足KKT条件，则中止
        if if_all_match_kkt is True:
            break
    # 计算支持向量epochs=100
    self.support_vectors = np.where(self.alpha > 1e-3)[0]
# 显示最终结果
```

结果预测及精确度检测

经过预测函数，将大于 0 的值设为 1 类（卡马类），将小于 0 的值设为-1 类（罗莎类）。

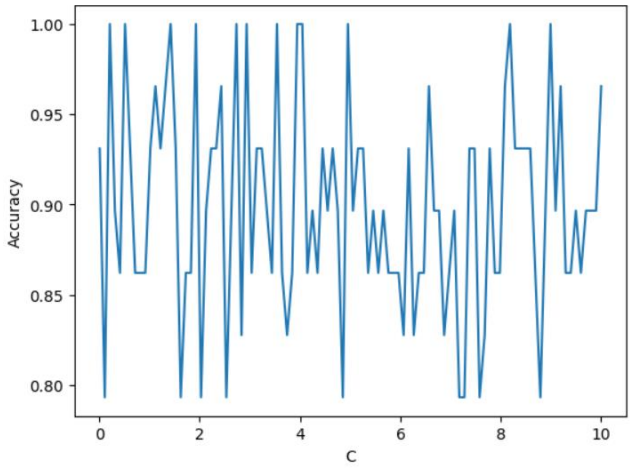
```
# 预测函数
def predict(self, x):
    return x.dot(self.w) + self.b
```

```
Z_test = svm.predict(X_test)
#print(Z_test)
#print(Y_test)
# 若预测结果与真实结果相差不到0.5，则认为分类正确
for i in range(Y_test.shape[0]):
    Z_test[i] = sign(Z_test[i])
    if (Z_test[i] == Y_test[i]):
        count += 1
Z_test = pd.DataFrame(Z_test)
Z_test.to_csv('result1.csv', encoding='utf_8_sig')
print(count / Y_test.shape[0])      #输出准确率
```

4. 模型表现

超参数 C 的选择

由于数据集的样本太少，因此很多 C 值的精确率都为 1，但是整体上精确度都比较高，因此为使模型较为合理，我选取 C =0.1，其中一张 C-精确率的结果图如下：



交叉验证

本文采用的是交叉验证法，将数据集分为五份，每份依次当作测试集，共进行五次训练，其中每次测试的表现如下表所示：

数据集	**1**	**2**	**3**	**4**	**5**
精确度（%）	**93.1**	**91.9**	**92.8**	**91.0**	**85.6**

由此计算得平均精确度为：90.88，效果还不错。

5. 总结

本次实验大作业，手动实现了 SVM 模型以及 SMO 算法，让我对这部分内容的理解更加深刻，同时实验结果也还不错，基本能够完成种子分类的任务。