第十五周学习笔记—手撕支持向量机代码

2022-07-22

第一章　SMO 算法

引言

面对这样的优化问题:

$$\min_{\alpha} \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha^{(i)}\alpha^{(j)}K(x^{(i)}\cdot x^{(j)}) - \sum_{i=1}^{m}\alpha^{(i)}$$

$$s.t.\ \sum_{i=1}^{m}\alpha^{(i)}y^{(i)} = 0\ ,\ 0 \le \alpha^{(i)} \le C\ ,\ i = 1, 2, \cdots, m$$

参数:α 是拉格朗日乘子构成的变量，有 $\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_N)^T$，每一个拉格朗日乘子对应一个样本点，例如:$\alpha_1 \to (x_1, y_1)$。

1. 坐标下降法

每次只完成一个参数的更新。

我们接下来举一个简单的例子来说明:

$$\arg\ \min_{x_1, x_2} f(x_1, x_2) = x_1^2 + 2x_2^2 - x_1x_2 + 1$$

①: 初始值 $(x_1^{(0)}, x_2^{(0)})^T$

②: 选择其中一个进行更新

例如选择 $x_1^{(0)}$，固定 $x_2^{(0)}$，使得问题转化为:

$$\arg\ \min_{x_1, x_2} f(x_1, x_2^{(0)})$$

我们采用费马原理如下:

$$\frac{\partial f}{\partial x_1} = 2x_1 - x_2^{(0)} = 0 \Rightarrow x_2 = \frac{x_2^0}{2}$$

③: 更新 x_2，固定 $x_1 = x_1^1$，求解 x_2

$$\arg \min_{x_1,x_2} f(x_1^{(1)}, x_2)$$

$$\frac{\partial f}{\partial x_2} = 4x_2 - x_1^{(1)} \Rightarrow x_2^{(1)} = \frac{x_1^1}{4}$$

④: 重复上面的①→③直到收敛为止。

坐标下降法可以应用在非线性支持向量机吗?

不妨选取 α_1，固定 $\alpha_2, \alpha_3, \cdots, \alpha_N$。

①: 初始值 $\alpha^{(0)} = (\alpha_1^{(0)}, \alpha_2^{(0)}, \cdots, \alpha_N^{(0)})$

②: 在固定 $\alpha_2, \alpha_3, \cdots, \alpha_N$ 下，求 α_1

使得:

$$\min_{\alpha} \ W(\alpha_1, \alpha_2^{(0)}, \alpha_3^{(0)}, \cdots, \alpha_N^{(0)})$$

$$s.t. \ \alpha_i y_i = -\sum_{i=2}^{N} \alpha_i^{(0)} y_i \ , \ 0 \le \alpha_i \le C$$

此时 α_1 可以直接由约束条件得到具体的值，无法进行更新。

所以坐标下降法用于非线性支持向量机的方法失败了。

但是我们可以换个思路，固定剩余的 $N-2$ 个变量，求两个变量，这就是接下来要讲的序列最小最优算法的最初想法。

2.SMO 算法

SMO 算法要解决如下问题:

$$\min_{\alpha} \ \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha^{(i)} \alpha^{(j)} K(x^{(i)} \cdot x^{(j)}) - \sum_{i=1}^{m} \alpha^{(i)}$$

$$\textbf{s.t.} \ \sum_{i=1}^{m} \alpha^{(i)} y^{(i)} = 0 \ , \ 0 \le \alpha^{(i)} \le C \ , \ i = 1, 2, \cdots, m$$

我们选择两个变量，α_1 , α_2，其他变量固定，于是 SMO 的最优化问题的子问题为:

$$\begin{aligned} \min_{\alpha_1,\alpha_2} W(\alpha^{(1)}, \alpha^{(2)}) = & \frac{1}{2} K_{11} {\alpha^{(1)}}^2 + \frac{1}{2} K_{22} {\alpha^{(2)}}^2 + y^{(1)} y^{(2)} K_{12} \alpha^{(1)} \alpha^{(2)} \\ & - (\alpha^{(1)} + \alpha^{(2)}) + y^{(1)} \alpha^{(1)} \sum_{i=3}^{m} y^{(i)} \alpha^{(i)} K_{i1} + y^{(2)} \alpha^{(2)} \sum_{i=3}^{m} y^{(i)} \alpha^{(i)} K_{i2} \end{aligned}$$

$$\textbf{s.t.} \qquad \alpha^{(1)} y^{(1)} + \alpha^{(2)} y^{(2)} = -\sum_{i=3}^{m} y^{(i)} \alpha^{(i)} = \zeta \ , \ 0 \le \alpha^{(i)} \le C \ , \ i = 1, 2$$

其中，$K_{ij} = K(x_i, x_j)$ ，$i, j = 1, 2, \cdots, N$，ζ 是常数，目标函数中省略了不含 $\alpha^{(1)}$, $\alpha^{(2)}$ 项。

为了叙述简单，记:

$$g(x) = \sum_{i=1}^{m} \alpha^{(i)} y^{(i)} K(x^{(i)}, x) + b$$

$$E_i = g(x^{(i)}) - y^{(i)} = (\sum_{j=1}^{m} \alpha^{(j)} y^{(j)} K(x^{(j)}, x^{(i)}) + b) - y^{(i)}$$

$$V_i = \sum_{j=3}^{m} \alpha^{(j)} y^{(j)} K(x^{(j)}, x^{(i)}) = g(x^{(i)}) - \sum_{j=1}^{2} \alpha^{(j)} y^{(j)} K(x^{(j)}, x^{(i)}) - b$$

目标函数可写成:

$$W(\alpha^{(1)}, \alpha^{(2)}) = \frac{1}{2} K_{11} {\alpha^{(1)}}^2 + \frac{1}{2} K_{22} {\alpha^{(2)}}^2 + y^{(1)} y^{(2)} K_{12} \alpha^{(1)} \alpha^{(2)} - (\alpha^{(1)} + \alpha^{(2)}) + y^{(1)} \alpha^{(1)} v_1 + y^{(2)} \alpha^{(2)} v_2$$

我们的表示方法如下:

$$K_{11} = K(x_1, x_2) \text{ , } K_{22} = K(x_2, x_2)$$

$$K_{12} = K(x_1, x_2) \text{ , } K_{1j} = K(x_1, x_j)$$

$$K_{2j} = K(x_2, x_j)$$

由 $\alpha^{(1)} y^{(1)} = \zeta - \alpha^{(2)} y^{(2)}$ 可将 $\alpha^{(1)}$ 表示为:

$$\alpha^{(1)} = (\zeta - \alpha^{(2)} y^{(2)}) y^{(1)}$$

且 ${y^{(i)}}^2 = 1$。

$$W(\alpha^{(2)}) = \frac{1}{2} K_{11} (\zeta - \alpha^{(2)} y^{(2)})^2 + \frac{1}{2} K_{22} {\alpha^{(2)}}^2 + y^{(2)} K_{12} (\zeta - \alpha^{(2)} y^{(2)}) \alpha^{(2)} - ((\zeta - \alpha^{(2)} y^{(2)}) y^{(1)} + \alpha^{(2)}) + (\zeta - \alpha^{(2)} y^{(2)}) v_1 + y^{(2)} \alpha^{(2)} v_2$$

对 $\alpha^{(2)}$ 求导:

$$\frac{\partial W}{\partial \alpha^{(2)}} = K_{11} \alpha^{(2)} + K_{22} \alpha^{(2)} - 2 K_{12} \alpha^{(2)} - K_{11} \zeta y^{(2)} + K_{12} \zeta y^{(2)} + y^{(1)} y^{(2)} - 1 - v_1 y^{(2)} + y^{(2)} v_2$$

令其为 0，得到:

$$(K_{11} + K_{22} - 2K_{12}) \alpha^{(2)} = y^{(2)} (y^{(2)} - y^{(1)} + \zeta K_{11} - \zeta K_{12} + v_1 - v_2)$$

$$= y^{(2)} (y^{(2)} - y^{(1)} + \zeta K_{11} - \zeta K_{12} + (g(x_1) - \sum_{j=1}^{2} \alpha^{(j)} y^{(j)} K_{1j} - b) - (g(x_2) - \sum_{j=1}^{2} \alpha^{(j)} y^{(j)} K_{2j} - b))$$

将 $\zeta = \alpha_{old}^{(1)} y^{(1)} + \alpha_{old}^{(2)} y^{(2)}$ 代入，得到:

$$\begin{aligned}(K_{11} + K_{22} - 2K_{12})\alpha_{new,unc}^{(2)} &= y^{(2)}((K_{11} + K_{22} - 2K_{12})\alpha_{old}^{(2)} y^{(2)} + y^{(2)} - y^{(1)} + g(x_1) - g(x_2)) \\ &= (K_{11} + K_{22} - 2K_{12})\alpha_{old}^{(2)} + y^{(2)}(E_1 - E_2)\end{aligned}$$

令 $\eta = K_{11} + K_{22} - 2K_{12}$ 代入，得到:

$$\alpha_{new,unc}^{(2)} = \alpha_{old}^{(2)} + \frac{y^{(2)}(E_1 - E_2)}{\eta}$$

上面的结果我们求得的是无约束的解，我们需要看一个经过约束条件后的迭代条件。条件如下:

$$\begin{cases} \alpha_1 y_1 + \alpha_2 y_2 = \zeta \\ 0 \le \alpha_1 \le C \\ 0 \le \alpha_2 \le C \end{cases}$$

我们分情况讨论:

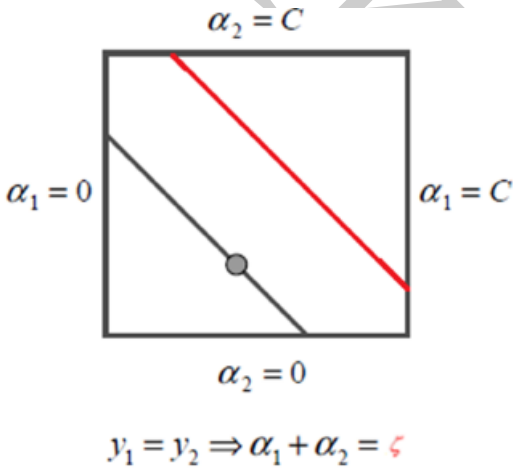
① $y_1 = y_2$:

$$\alpha_1 + \alpha_2 = y_1 \zeta = k$$

② $y_1 \neq y_2$:

$$\alpha_1 - \alpha_2 = y_1 \zeta = k$$

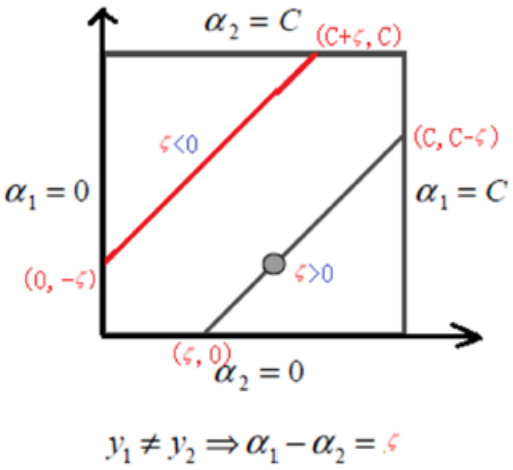
我们对于第一种情况:



此时的区间为:

$$L = \max(0, \zeta - C) = \max(0, \alpha_{old}^{(2)} + \alpha_{old}^{(1)} - C)$$
$$H = \min(C, \zeta) = \min(C, \alpha_{old}^{(2)} + \alpha_{old}^{(1)})$$

我们对于第二种情况:



此时的区间为:

$$L = \max(0, -\zeta) = \max(0, \alpha_{old}^{(2)} - \alpha_{old}^{(1)})$$
$$H = \min(C, C - \zeta) = \min(C, C + \alpha_{old}^{(2)} - \alpha_{old}^{(1)})$$

我们的 $\alpha^{(2)}$ 的区间为:

$$L \leq \alpha^{(2)} \leq H$$

因此我们得到的最终的 $\alpha^{(2)}$ 的解为:

$$\alpha_{new}^{(2)} = \begin{cases} H\ ,\ \alpha_{new,unc}^{(2)} > H \\ \alpha_{new,unc}^{(2)}\ ,\ L \leq \alpha_{new,unc}^{(2)} \leq H \\ L\ ,\ \alpha_{new,unc}^{(2)} < L \end{cases}$$

求得 $\alpha_{new}^{(1)}$ 为:

$$\alpha_{new}^{(1)} = \alpha_{old}^{(1)} + y^{(1)}y^{(2)}(\alpha_{old}^{(2)} - \alpha_{new}^{(2)})$$

我们接下来是计算 b 值，我们有:

$$g(x) = \sum_{i=1}^{m} \alpha^{(i)}y^{(i)}K(x^{(i)}, x) + b$$
$$E_i = g(x^{(i)}) - y^{(i)} = (\sum_{j=1}^{m} \alpha^{(j)}y^{(j)}K(x^{(j)}, x^{(i)}) + b) - y^{(i)}$$

解. (1) 当 $0 < \alpha_{new}^{(1)} < C$ 时，有:

$$\sum_{i=1}^{m} y^{(i)}\alpha^{(i)}K_{i1} + b = y^{(1)}$$

因此:

$$b_{new}^{(1)} = y^{(1)} - \sum_{i=3}^{m} y^{(i)}\alpha^{(i)}K_{i1} - \alpha_{new}^{(1)}y^{(1)}K_{11} - \alpha_{new}^{(2)}y^{(2)}K_{21}$$

由 E_1 定义可知:

$$E_1 = \sum_{i=3}^{m} y^{(i)}\alpha^{(i)}K_{i1} + \alpha_{old}^{(1)}y^{(1)}K_{11} + \alpha_{old}^{(2)}y^{(2)}K_{21} + b_{old} - y^{(1)}$$

变形得:

$$y^{(1)} - \sum_{i=3}^{m} y^{(i)}\alpha^{(i)}K_{i1} = -E_1 + \alpha_{old}^{(1)}y^{(1)}K_{11} + \alpha_{old}^{(2)}y^{(2)}K_{21} + b_{old}$$

代入 $b_{new}^{(1)} = y^{(1)} - \sum_{i=3}^{m} y^{(i)}\alpha^{(i)}K_{i1} - \alpha_{new}^{(1)}y^{(1)}K_{11} - \alpha_{new}^{(2)}y^{(2)}K_{21}$ 得:

$$b_{new}^{(1)} = -E_1 - y^{(1)}K_{11}(\alpha_{new}^{(1)} - \alpha_{old}^{(1)}) - y^{(2)}K_{21}(\alpha_{new}^{(2)} - \alpha_{old}^{(2)}) + b_{old}$$

(2) 同理若 $0 < \alpha_{new}^{(2)} < C$，可得:

$$b_{new}^{(2)} = -E_2 - y^{(1)}K_{12}(\alpha_{new}^{(1)} - \alpha_{old}^{(1)}) - y^{(2)}K_{22}(\alpha_{new}^{(2)} - \alpha_{old}^{(2)}) + b_{old}$$

(3) 若 $\alpha_{new}^{(1)}$ 和 $\alpha_{new}^{(2)}$ 同时满足 $0 < \alpha_{new}^{(i)} < C$，则:

$$b_{new}^{(1)} = b_{new}^{(2)}$$

若 $\alpha_{new}^{(1)}$ 和 $\alpha_{new}^{(2)}$ 是 0 或者 C，则:

$$b_{new} = \frac{b_{new}^{(1)} + b_{new}^{(2)}}{2}$$

□

3.SMO 算法推导结果

$$g(x) = \sum_{i=1}^{m} \alpha^{(i)}y^{(i)}K(x^{(i)}, x) + b$$

$$E_i = g(x^{(i)}) - y^{(i)} = (\sum_{j=1}^{m} \alpha^{(j)}y^{(j)}K(x^{(j)}, x^{(i)}) + b) - y^{(i)}$$

$$\eta = K_{11} + K_{22} - 2K_{12}$$

$$\alpha_{new,unc}^{(2)} = \alpha_{old}^{(2)} + \frac{y^{(2)}(E_1 - E_2)}{\eta}$$

若 $y^{(1)} \neq y^{(2)}$:

$$L = \max(0, -\zeta) = \max(0, \alpha_{old}^{(2)} - \alpha_{old}^{(1)})$$
$$H = \min(C, C - \zeta) = \min(C, C + \alpha_{old}^{(2)} - \alpha_{old}^{(1)})$$

若 $y^{(1)} = y^{(2)}$:

$$L = \max(0, \zeta - C) = \max(0, \alpha_{old}^{(2)} + \alpha_{old}^{(1)} - C)$$
$$H = \min(C, \zeta) = \min(C, \alpha_{old}^{(2)} + \alpha_{old}^{(1)})$$

$$\alpha_{new}^{(2)} = \begin{cases} H \ , & \alpha_{new,unc}^{(2)} > H \\ \alpha_{new,unc}^{(2)} \ , & L \leq \alpha_{new,unc}^{(2)} \leq H \\ L \ , & \alpha_{new,unc}^{(2)} < L \end{cases}$$

$$\alpha_{new}^{(1)} = \alpha_{old}^{(1)} + y^{(1)} y^{(2)} (\alpha_{old}^{(2)} - \alpha_{new}^{(2)})$$
$$b_{new}^{(1)} = -E_1 - y^{(1)} K_{11} (\alpha_{new}^{(1)} - \alpha_{old}^{(1)}) - y^{(2)} K_{21} (\alpha_{new}^{(2)} - \alpha_{old}^{(2)})$$
$$b_{new}^{(2)} = -E_2 - y^{(1)} K_{12} (\alpha_{new}^{(1)} - \alpha_{old}^{(1)}) - y^{(2)} K_{22} (\alpha_{new}^{(2)} - \alpha_{old}^{(2)})$$

若 $0 < \alpha_{new}^{(1)} < C$，则:

$$b = b_{new}^{(1)}$$

若 $0 < \alpha_{new}^{(2)} < C$，则:

$$b = b_{new}^{(2)}$$

其他情况:

$$b_{new} = \frac{b_{new}^{(1)} + b_{new}^{(2)}}{2}$$

第二章　SVM 的代码实现

引言:Platt 的 SMO 算法

1996 年，John Platt 发布了一个称为 SMO 的强大算法，用于训练 SVM。

SMO 表示序列最小优化。Platt 的 SMO 算法是将大优化问题分解为多个小优化问题来求解的。这些小优化问题往往很容易求解，并且对它们进行顺序求解的结果与将它们作为整体来求解的结果是完全一致的。在结果完全相同的同时，SMO 算法的求解时间短很多。

SMO 算法的目标是求出一系列 α 和 b，一旦求出了这些 α，就很容易计算出权重向量 ω 并得到分隔超平面。

SMO 算法的工作原理是：每次循环中选择两个 α 进行优化处理。一旦找到一对合适的 α，那么就增大其中一个同时减小另一个。这里所谓的“合适”就是指两个 α 必须要符合一定的条件，条件之一就是这两个 α 必须要在间隔边界之外，而其第二个条件则是这两个 α 还没有进行过区间化处理或者不在边界上。

1. 应用简化版 SMO 算法处理小规模数据集

首先在数据集上遍历每一个 α，然后在剩下的 α 集合中随机选择另一个 α，从而构建 α 对。这里有一点相当重要，就是我们要同时改变两个 α。之所以这样做是因为我们有一个约束条件：

$$\sum_{i=1}^{N} \alpha_i y_i = 0$$

此时我们使用的是线性支持向量机。

为此，我们将构建一个辅助函数，用于在某个区间范围内随机选择一个整数。同时，我们也需要另一个辅助函数，用于在数值太大时对其进行调整。下面给出了这两个函数的实现:

```
"""
函数说明: 读取数据,处理文本数据
Parameters:
    file_name - 文件名
Returns:
    data_mat - 数据矩阵
    label_mat - 数据标签
"""
def load_dataset(file_name):
    # 数据矩阵
    data_mat = []
    # 标签向量
    label_mat = []
    # 打开文件
    fr = open(file_name)
    # 逐行读取
    for line in fr.readlines():
        # 去掉每一行首尾的空白符, 例如'\n','\r','\t',' '
        # 将每一行内容根据'\t'符进行切片
        line_array = line.strip().split('\t')
        # 添加数据(100个元素排成一行)
        data_mat.append([float(line_array[0]), float(line_array[1])])
        # 添加标签(100个元素排成一行)
        label_mat.append(int(line_array[2]))
    return data_mat, label_mat
```

```
"""
函数说明: 随机选择alpha_j。alpha的选取, 随机选择一个不等于i值的j
Parameters:
    i - 第一个alpha的下标
    m - alpha参数个数
Returns:
    j - 返回选定的数字
"""
def select_J_rand(i, m):
    """
    :param i: 表示alpha_i
    :param m: 表示样本数
```

```
    :return:
    """
    j = i
    while(j == i):
        # 如果i和j相等，那么就从样本中随机选取一个，可以认为j就是选择的
                                                alpha_2
        # uniform()方法将随机生成一个实数，它在[x, y)范围内
        j = int(random.uniform(0, m))
    return j
```

```
"""
函数说明：修剪alpha
Parameters:
    a_j - alpha值
    H - alpha上限
    L - alpha下限
Returns:
    a_j - alpha值
用于调整大于H或小于L的alpha值。
"""
def clip_alpha(a_j, H, L):
    if a_j > H:
        a_j = H
    if L > a_j:
        a_j = L
    return a_j
```

第一个函数就是我们所熟知的 load-dataset() 函数，该函数打开文件并对其进行逐行解析，从而得到每行的类标签和整个数据矩阵。

下一个函数 select-J-rand() 有两个参数值，其中 i 是第一个 α 的下标，m 是所有 α 的数目。只要函数值不等于输入值 i，函数就会进行随机选择。

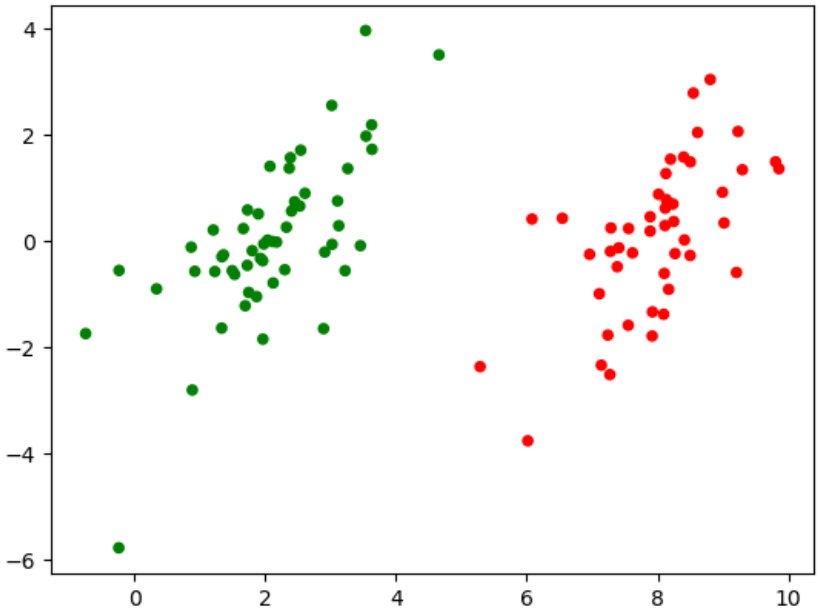
最后一个辅助函数就是 clip-alpha()，它是用于调整大于 H 或小于 L 的 α 值。

```
数据标签为：
 [-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, 1,
                                        1, 1, -1, -1, -1, -1, -1, -1, -1]
```

可以看得出来，这里采用的类别标签是 1 和 1，而不是 0 和 1。

我们可视化一下自己的数据集:

```
fig = plt.figure()
ax = fig.add_subplot(111)
cm = mpl.colors.ListedColormap(['g', 'r'])
ax.scatter(np.array(data_mat)[:, 0], np.array(data_mat)[:, 1], c=np.array(
                                        label_mat), cmap=cm, s=20)
```



我们的伪代码大致如下:

```
创建一个alpha向量并将其初始化为0向量

当迭代次数小于最大迭代次数时（外循环）

    对数据集中的每个数据向量（内循环）:

    如果该数据向量可以被优化:

        随机选择另外一个数据向量

        同时优化这两个向量

        如果两个向量都不能被优化，退出内循环
```

```
如果所有向量都没有被优化，增加迭代数目，继续下一次循环
```

下面的代码是 SMO 算法的一个简单版本:

```
"""
函数说明：简化版SMO算法
Parameters:
    data_mat_in - 数据矩阵
    class_labels - 数据标签
    C - 松弛变量
    toler - 容错率
    max_iter - 最大迭代次数
Returns:
    None
"""
def smo_simple(data_mat_in, class_labels, C, toler, max_iter):
    """
    :param data_mat_in: 相当于我们的x
    :param class_labels: 相当于我们的y
    :param C: 惩罚因子
    :param toler:
    :param max_iter:
    :return:
    """
    # 转换为numpy的mat矩阵存储(100,2)
    data_matrix = np.mat(data_mat_in) #相当于x
    # 转换为numpy的mat矩阵存储并转置(100,1)
    label_mat = np.mat(class_labels).transpose() #相当于y
    # 初始化b参数，统计data_matrix的维度,m:行; n:列
    b = 0
    # 统计dataMatrix的维度,m:100行; n:2列
    m, n = np.shape(data_matrix)
    # 初始化alpha参数，设为0
    alphas = np.mat(np.zeros((m, 1)))
    # 初始化迭代次数
    iter_num = 0
    # 最多迭代maxIter次
    while(iter_num < max_iter):
        alpha_pairs_changed = 0
        for i in range(m):
```

```
            # 步骤1: 计算误差Ei
            # multiply(a,b)就是个乘法，如果a,b是两个数组，那么对应元素相乘
            # .T为转置，转置的目的是因为后面的核函数是一个向量。核函数我们
                                                    最原始的方式，可以直接使
                                                    用点乘就可以，即x_i.x
            fxi = float(np.multiply(alphas, label_mat).T * (data_matrix *
                                                    data_matrix[i, :].T)) + b
            # 误差项计算公式
            Ei = fxi - float(label_mat[i])
            # 优化alpha，设定一定的容错率
            if((label_mat[i] * Ei < -toler) and (alphas[i] < C)) or ((
                                                    label_mat[i] * Ei > toler
                                                    ) and (alphas[i] > 0)):
                # 随机选择另一个alpha_i成对比优化的alpha_j
                j = select_J_rand(i, m)
                # 步骤1，计算误差Ej
                fxj = float(np.multiply(alphas, label_mat).T * (data_matrix
                                                     * data_matrix[j, :].
                                                    T)) + b
                # 误差项计算公式
                Ej = fxj - float(label_mat[j])
                # 保存更新前的alpha值，使用深拷贝(完全拷贝)A深层拷贝为B，A
                                                    和B是两个独立的个体
                alpha_i_old = alphas[i].copy()
                alpha_j_old = alphas[j].copy()
                # 步骤2: 计算上下界H和L
                if(label_mat[i] != label_mat[j]):
                    L = max(0, alphas[j]-alphas[i])
                    H = min(C, C + alphas[j] - alphas[i])
                else:
                    L = max(0, alphas[j] + alphas[i] - C)
                    H = min(C, alphas[j] + alphas[i])
                if(L == H):
                    print("L == H")
                    continue
                # 步骤3: 计算eta，转置表示相乘没有错误，相当于求的-eta
                eta = 2.0 * data_matrix[i, :] * data_matrix[j, :].T -
                                                    data_matrix[i, :] *
                                                    data_matrix[i, :].T -
```

```
                                                data_matrix[j, :] *
                                               data_matrix[j, :].T
            if eta >= 0:
                print("eta>=0")
                continue
            # 步骤4: 更新alpha_j
            alphas[j] -= label_mat[j] * (Ei - Ej) / eta
            # 步骤5: 修剪alpha_j
            alphas[j] = clip_alpha(alphas[j], H, L)
            if(abs(alphas[j] - alpha_j_old) < 0.00001):
                print("alpha_j变化太小")
                continue
            # 步骤6: 更新alpha_i
            alphas[i] += label_mat[j] * label_mat[i] * (alpha_j_old -
                                               alphas[j])
            # 步骤7: 更新b_1和b_2
            b1 = b - Ei - label_mat[i] * (alphas[i] - alpha_i_old) *
                                               data_matrix[i, :] *
                                               data_matrix[i, :].T -
                                                label_mat[j] * (
                                               alphas[j] -
                                               alpha_j_old) *
                                               data_matrix[j, :] *
                                               data_matrix[i, :].T
            b2 = b - Ej - label_mat[i] * (alphas[i] - alpha_i_old) *
                                               data_matrix[i, :] *
                                               data_matrix[j, :].T -
                                                label_mat[j] * (
                                               alphas[j] -
                                               alpha_j_old) *
                                               data_matrix[j, :] *
                                               data_matrix[j, :].T
            # 步骤8: 根据b_1和b_2更新b
            if(0 < alphas[i] < C):
                b = b1
            elif(0 < alphas[j] < C):
                b = b2
            else:
                b = (b1 + b2) / 2.0
```

```
                # 统计优化次数
                alpha_pairs_changed += 1
                # 打印统计信息
                print("第%d次迭代 样本: %d, alpha优化次数: %d" % (iter_num
                                                        , i,
                                                        alpha_pairs_changed))
        # 更新迭代次数
        if(alpha_pairs_changed == 0):
            iter_num += 1
        else:
            iter_num = 0
        print("迭代次数: %d" % iter_num)
    return b, alphas
```

该函数有 5 个输入参数，分别是：数据集、类别标签、常数 C、容错率和退出前最大的循环次数。

上述函数将多个列表和输入参数转换成 NumPy 矩阵，这样就可以简化很多数学处理操作。由于转置了类别标签，因此我们得到的就是一个列向量而不是列表。于是类别标签向量的每行元素都和数据矩阵中的行一一对应。

我们也可以通过矩阵 data-mat-in 的 shape 属性得到常数 m 和 n。最后，我们就可以构建一个 alpha 列矩阵，矩阵中元素都初始化为 0，并建立一个 iter 变量。该变量存储的则是在没有任何 alpha 改变的情况下遍历数据集的次数。当该变量达到输入值 max-iter 时，函数结束运行并退出。

每次循环当中，将 alpha-pairs-changed 先设为 0，然后再对整个集合顺序遍历。变量 alpha-pairs-changed 用于记录 alpha 是否已经进行优化。当然，在循环结束时就会得知这一点。

首先，fXi 能够计算出来，这就是我们预测的类别。然后，基于这个实例的预测结果和真实结果的比对，就可以计算误差 Ei。如果误差很大，那么可以对该数据实例所对应的 alpha 值进行优化。

在 if 语句中，不管是正间隔还是负间隔都会被测试。并且在该 if 语句中，也要同时检查 alpha 值，以保证其不能等于 0 或 C。由于后面 alpha 小于 0 或大于 C 时将被调整为 0 或 C，所以一旦在该 if 语句中它们等于这两个值的话，那么它们就已经在“边界”上了，因而不再能够减小或增大，因此也就不值得再对它们进行优化了。

接下来，可以利用程序中的辅助函数来随机选择第二个 alpha 值，即 alpha[j] 。

同样，可以采用第一个 alpha 值（alpha[i]）的误差计算方法，来计算这个 alpha 值

的误差。

这个过程可以通过 copy() 的方法来实现，因此稍后可以将新的 alpha 值与老的 alpha 值进行比较。Python 则会通过引用的方式传递所有列表，所以必须明确地告知 Python 要为 alpha-i-old 和 alpha-j-old 分配新的内存；否则的话，在对新值和旧值进行比较时，我们就看不到新旧值的变化。

之后我们开始计算 L 和 H ，它们用于将 alpha[j] 调整到 0 到 C 之间。如果 L 和 H 相等，就不做任何改变，直接执行 continue 语句。这在 Python 中，则意味着本次循环结束直接运行下一次 for 的循环。

Eta 是 alpha[j] 的最优修改量，在那个很长的计算代码行中得到。如果 eta 为 0，那就是说需要退出 for 循环的当前迭代过程。该过程对真实 SMO 算法进行了简化处理。如果 eta 为 0，那么计算新的 alpha[j] 就比较麻烦了，这里我们就不对此进行详细的介绍了。有需要的可以阅读 Platt 的原文来了解更多的细节。现实中，这种情况并不常发生，因此忽略这一部分通常也无伤大雅。

于是，可以计算出一个新的 alpha[j]，然后利用程序中的辅助函数以及 L 与 H 值对其进行调整。

然后，就是需要检查 alpha[j] 是否有轻微改变。如果是的话，就退出 for 循环。然后，alpha[i] 和 alpha[j] 同样进行改变，虽然改变的大小一样，但是改变的方向正好相反（即如果一个增加，那么另外一个减少）。在对 alpha[i] 和 alpha[j] 进行优化之后，给这两个 alpha 值设置一个常数项 b 。

最后，在优化过程结束的同时，必须确保在合适的时机结束循环。如果程序执行到 for 循环的最后一行都不执行 continue 语句，那么就已经成功地改变了一对 alpha，同时可以增加 alpha-pairs-changed 的值。在 for 循环之外，需要检查 alpha 值是否做了更新，如果有更新则将 iter 设为 0 后继续运行程序。只有在所有数据集上遍历 max-iter 次，且不再发生任何 alpha 修改之后，程序才会停止并退出 while 循环。

我们得到的计算 ω 的程序如下:

```
"""
函数说明：计算w
Returns:
    data_mat - 数据矩阵
    label_mat - 数据标签
    alphas - alphas值
Returns:
    w - 直线法向量
"""
```

```
def get_w(data_mat_in, class_labels, alphas):
    """
    :param data_mat: x
    :param label_mat: y
    :param alphas:
    :return:
    """
    data_mat=np.mat(data_mat_in)
    label_mat=np.mat(class_labels).transpose()
    m,n=np.shape(data_mat)
    # 初始化w都为1
    w=np.zeros((n,1))
    # dot()函数是矩阵乘，而*则表示逐个元素相乘
    # w = sum(alpha_i * yi * xi)
    #循环计算
    for i in range(m):
        w+=np.multiply(alphas[i]*label_mat[i],data_mat[i,:].T)
    return w
```

我们得到参数的值分别为:

```
b, alphas = smo_simple(data_mat, label_mat, 0.6, 0.001, 40)
w = get_w(data_mat, label_mat, alphas)
print('b=',b)
print('w=',w)
print('alphas=',alphas[alphas>0])
```

```
b= [[-3.80302032]]
w= [[ 0.81089037]
 [-0.27777357]]
alphas= [[0.15706497 0.14445528 0.06542831 0.36694855]]
```

我们的迭代过程如下所示:

```
第0次迭代 样本: 0, alpha优化次数: 1
L == H
alpha_j变化太小
alpha_j变化太小
第0次迭代 样本: 8, alpha优化次数: 2
L == H
L == H
```

```
L == H
alpha_j变化太小
L == H
alpha_j变化太小
第0次迭代 样本：26， alpha优化次数：3
L == H
L == H
......
迭代次数：37
alpha_j变化太小
alpha_j变化太小
迭代次数：38
alpha_j变化太小
alpha_j变化太小
迭代次数：39
alpha_j变化太小
alpha_j变化太小
迭代次数：40
```

为了解哪些数据点是支持向量，输入：

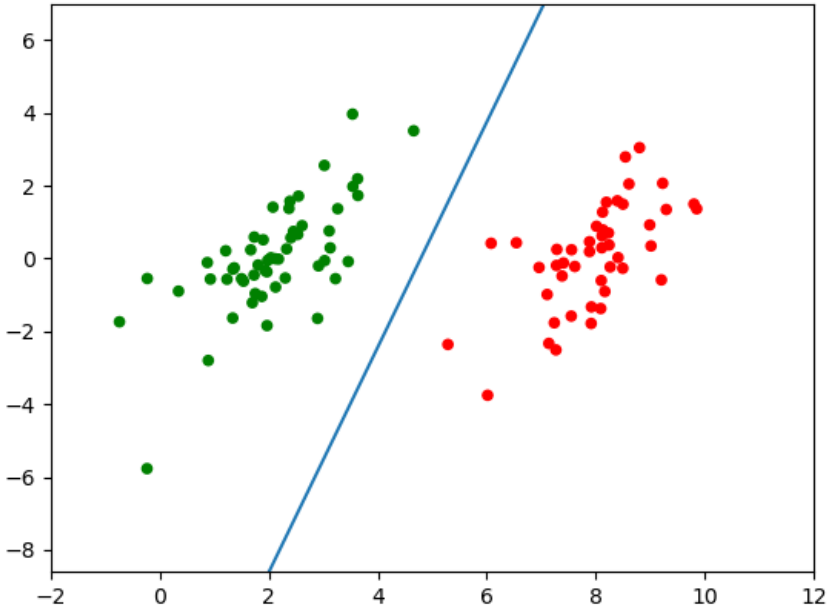
```
for i in range(100):
    if alphas[i]>0.0:
        print(data_mat[i],label_mat[i])
```

得到的结果如下：

```
[4.658191, 3.507396] -1
[3.457096, -0.082216] -1
[2.893743, -1.643468] -1
[6.080573, 0.418886] 1
```

因为是线性可分，我们画出决策边界:

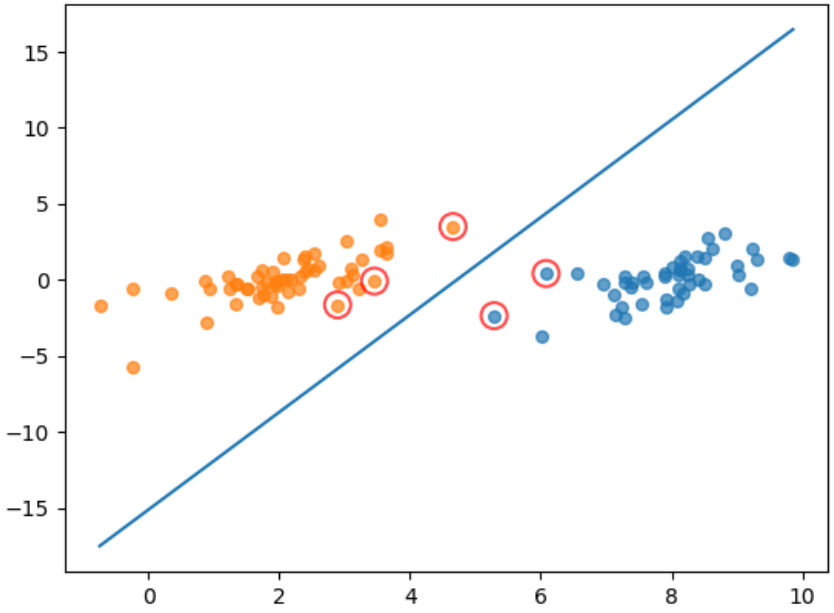
```
x = np.arange(-2.0, 12, 0.1)
# w0x0+w1x1=0,x1=-w0x0/w1
y = (-w[0] * x - b) / w[1]
ax.plot(x,y.reshape(-1,1))
ax.axis([-2,12,-8.6,7])
plt.show()
```



我们希望同时找出支持向量，下面是书中记录的案例:

```
"""
函数说明：分类结果可视化
Returns:
    dataMat - 数据矩阵
    w - 直线法向量
    b - 直线截距
Returns:
    None
"""
def show_classifer(data_mat, w, b):
    # 正样本
    data_plus = []
    # 负样本
    data_minus = []
    for i in range(len(data_mat)):
        if label_mat[i] > 0:
            data_plus.append(data_mat[i])
        else:
            data_minus.append(data_mat[i])
    # 转换为numpy矩阵
    data_plus_np = np.array(data_plus)
```

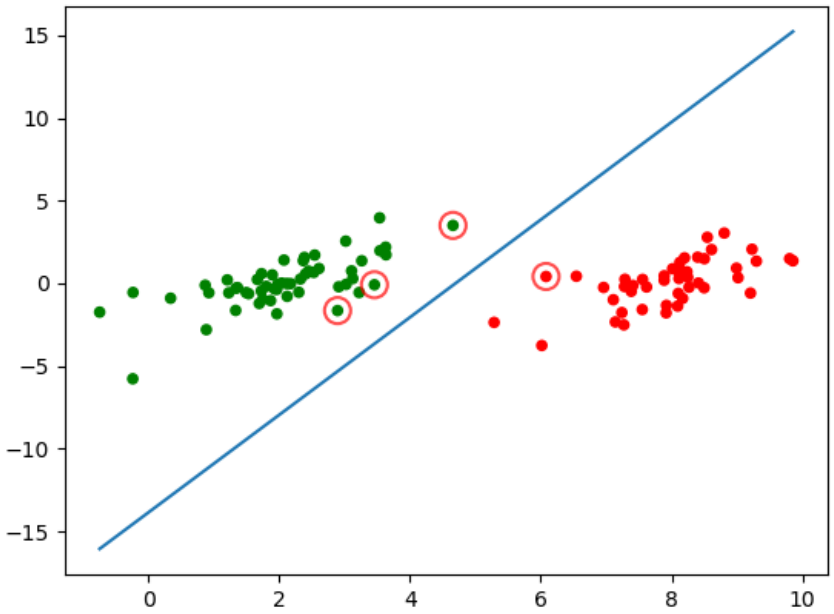
```
    # 转换为numpy矩阵
    data_minus_np = np.array(data_minus)
    # 正样本散点图（scatter）
    # transpose转置
    plt.scatter(np.transpose(data_plus_np)[0], np.transpose(data_plus_np)[1
                                            ], s=30, alpha=0.7)
    # 负样本散点图（scatter）
    plt.scatter(np.transpose(data_minus_np)[0], np.transpose(data_minus_np)
                                            [1], s=30, alpha=0.7)
    # 绘制直线
    x1 = max(data_mat)[0]
    x2 = min(data_mat)[0]
    a1, a2 = w
    b = float(b)
    a1 = float(a1[0])
    a2 = float(a2[0])
    y1, y2 = (-b - a1 * x1) / a2, (-b - a1 * x2) / a2
    plt.plot([x1, x2], [y1, y2])
    # 找出支持向量点
    # enumerate在字典上是枚举、列举的意思
    for i, alpha in enumerate(alphas):
        # 支持向量机的点
        if(abs(alpha) > 0):
            x, y = data_mat[i]
            plt.scatter([x], [y], s=150, c='none', alpha=0.7, linewidth=1.5
                                                  , edgecolors='red')
    plt.show()
```



我们换一种简单的写法:

```
def show_classifer(data_mat, w, b):
    data_mat, label_mat = load_dataset('testSet.txt')
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cm = mpl.colors.ListedColormap(['g', 'r'])
    ax.scatter(np.array(data_mat)[:, 0], np.array(data_mat)[:, 1], c=np.
                                            array(label_mat), cmap=cm, s=20)
    # 绘制直线
    x1 = max(data_mat)[0]
    x2 = min(data_mat)[0]
    a1, a2 = w
    b = float(b)
    a1 = float(a1[0])
    a2 = float(a2[0])
    y1, y2 = (-b - a1 * x1) / a2, (-b - a1 * x2) / a2
    plt.plot([x1, x2], [y1, y2])
    # 找出支持向量点
    # enumerate在字典上是枚举、列举的意思
    for i, alpha in enumerate(alphas):
        # 支持向量机的点
        if(abs(alpha) > 0):
```

```
            x, y = data_mat[i]
            plt.scatter([x], [y], s=150, c='none', alpha=0.7, linewidth=1.5
                                                    , edgecolors='red')
    plt.show()
```



在更大的数据集上，收敛时间会变得更长。在下一节中，我们优化算法，以提高算法的运行效率。

2. 完整版的 SMO 算法

在几百个点组成的小规模数据集上，简化版 SMO 算法的运行是没有什么问题的，但是在更大的数据集上的运行速度就会变慢。

刚才已经讨论了简化版 SMO 算法，下面我们就讨论完整版的 Platt SMO 算法。

在这两个版本中，实现 alpha 的更改和代数运算的优化环节一模一样。在优化过程中，唯一的不同就是选择 alpha 的方式。

完整版的 Platt SMO 算法应用了一些能够提速的启发方法。上一节的例子在执行时存在一定的时间提升空间。

Platt SMO 算法是通过一个外循环来选择第一个 alpha 值的，并且其选择过程会在两种方式之间进行交替：一种方式是在所有数据集上进行单遍扫描，另一种方式则是在非边界 alpha 中实现单遍扫描。

而所谓非边界 alpha 指的就是那些不等于边界 0 或 C 的 alpha 值。

对整个数据集的扫描相当容易，而实现非边界 alpha 值的扫描时，首先需要建立这些 alpha 值的列表，然后再对这个表进行遍历。同时，该步骤会跳过那些已知的不会改变的 alpha 值。

在选择第一个 alpha 值后，算法会通过一个内循环来选择第二个 alpha 值。在优化过程中，会通过最大化步长的方式来获得第二个 alpha 值。

在简化版 SMO 算法中，我们会在选择 j 之后计算错误率 Ej。但在这里，我们会建立一个全局的缓存用于保存误差值，并从中选择使得步长或者说 Ei-Ej 最大的 alpha 值。

辅助函数和上面的步骤一样，即下面的三个函数:

```
def load_dataset(file_name):
def select_j_random(i, m):
def clip_alpha(aj, H, L):
```

我们首先定义一个数据结构:

```
"""
类说明：维护所有需要操作的值
Parameters:
    dataMatIn - 数据矩阵
    classLabels - 数据标签
    C - 松弛变量
    toler - 容错率
Returns:
    None
"""
"""
定义一个新的数据结构
"""
class opt_struct:
    def __init__(self, data_mat_in, class_labels, C, toler):
        # 数据矩阵
        self.X = data_mat_in #传进来的数据
        # 数据标签
        self.label_mat = class_labels
        # 松弛变量
        self.C = C
        # 容错率
        self.tol = toler
```

```
        # 矩阵的行数
        self.m = np.shape(data_mat_in)[0]
        # 根据矩阵行数初始化alphas矩阵，一个m行1列的全零列向量
        self.alphas = np.mat(np.zeros((self.m, 1)))
        # 初始化b参数为0
        self.b = 0
        # 根据矩阵行数初始化误差缓存矩阵，第一列为是否有效标志位，其中0无
                                                效，1有效;第二列为实际的误差
                                                Ei的值
        """
        我们之前的定义为:Ei=gxi-yi
        yi是标签的实际值。
        gx=alpha_i*y_i*x_i.x,就相当于是w.x+b
        因为误差值经常用到，所以希望每次计算后放到一个缓存当中，将ecache一
                                                分为二，第一列是标志位，取值
                                                为0或者1，为1时表示已经算出来
        """
        self.ecache = np.mat(np.zeros((self.m, 2)))
```

首要的事情就是建立一个数据结构来保存所有的重要值，而这个过程可以通过一个对象来完成。只是作为一个数据结构来使用对象。

在将值传给函数时，我们可以通过将所有数据移到一个结构中来实现，这样就可以省掉手工输入的麻烦了。而此时，数据就可以通过一个对象来进行传递。实际上，当完成其实现时，可以很容易通过 Python 的字典来完成。

该方法可以实现其成员变量的填充。除了增加了一个 m×2 的矩阵成员变量 ecache 之外，这些做法和简化版 SMO 一模一样。ecache 的第一列给出的是 ecache 是否有效的标志位，而第二列给出的是实际的 E 值。

对于给定的 alpha 值，第一个辅助函数 cal-Ek() 能够计算 E 值并返回。以前，该过程是采用内嵌的方式来完成的，但是由于该过程在这个版本的 SMO 算法中出现频繁，这里必须要将其单独拎出来。

```
"""
函数说明：计算误差
Parameters:
    os - 数据结构
    k - 标号为k的数据
Returns:
    Ek - 标号为k的数据误差
```

```
"""
def cal_Ek(os, k):
    # multiply(a,b)就是个乘法，如果a,b是两个数组，那么对应元素相乘
    # .T为转置
    fXk = float(np.multiply(os.alphas, os.label_mat).T * (os.X * os.X[k, :]
                                                          .T) + os.b)
    # 计算误差项
    Ek = fXk - float(os.label_mat[k])
    # 返回误差项
    return Ek
```

函数 select-j() 用于选择第二个 alpha 或者说内循环的 alpha 值。回想一下，这里的目标是选择合适的第二个 alpha 值以保证在每次优化中采用最大步长。该函数的误差值与第一个 alpha 值 Ei 和下标 i 有关。

首先将输入值 Ei 在缓存中设置成为有效的。这里的有效意味着它已经计算好了。在 ecache 中，代码 nonzero(os.ecache[:,0].A)[0] 构建出了一个非零表。

NumPy 函数 nonzero() 返回了一个列表，而这个列表中包含以输入列表为目录的列表值，当然这里的值并非零。nonzero() 语句返回的是非零 E 值所对应的 alpha 值，而不是 E 值本身。

程序会在所有的值上进行循环并选择其中使得改变最大的那个值。如果这是第一次循环的话，那么就随机选择一个 alpha 值。当然也存在有许多更复杂的方式来处理第一次循环的情况，而上述做法就能够满足我们的目的。

```
"""
函数说明：内循环启发方式2
选择第二个待优化的alpha_j，选择一个误差最大的alpha_j
即，我们在选择alpha_2的时候做了改进，选择误差最大的
Parameters:
    i - 标号为i的数据的索引值
    oS - 数据结构
    Ei - 标号为i的数据误差
Returns:
    j - 标号为j的数据的索引值
    maxK - 标号为maxK的数据的索引值
    Ej - 标号为j的数据误差
"""
def select_j(i, os, Ei):
    # 初始化
```

```
    max_K = -1 #下标的索引值
    max_delta_E = 0
    Ej = 0
    # 根据Ei更新误差缓存，即先计算alpha_1以及E1值
    os.ecache[i] = [1, Ei] #放入缓存当中，设为有效
    # 对一个矩阵.A转换为Array类型
    # 返回误差不为0的数据的索引值
    valid_ecache_list = np.nonzero(os.ecache[:, 0].A)[0] #找出缓存中不为0
    # 有不为0的误差
    if(len(valid_ecache_list) > 1):
        # 遍历，找到最大的Ek
        for k in valid_ecache_list: #迭代所有有效的缓存，找到误差最大的E
            # 不计算k==i节省时间
            if k == i: #不选择和i相等的值
                continue
            # 计算Ek
            Ek = cal_Ek(os, k)
            # 计算|Ei - Ek|
            delta_E = abs(Ei - Ek)
            # 找到maxDeltaE
            if(delta_E > max_delta_E):
                max_K = k
                max_delta_E = delta_E
                Ej = Ek
        # 返回max_K, Ej
        return max_K, Ej #这样我们就得到误差最大的索引值和误差最大的值
    # 没有不为0的误差
    else: #第一次循环时是没有有效的缓存值的，所以随机选一个（仅会执行一次）
        # 随机选择alpha_j的索引值
        j = select_j_random(i, os.m)
        # 计算Ej
        Ej = cal_Ek(os, j)
    # 返回j, Ej
    return j, Ej
```

最后一个辅助函数是 update-Ek()，它会计算误差值并存入缓存当中。在对 alpha 值进行优化之后会用到这个值。

```
"""
函数说明：计算Ek,并更新误差缓存
```

```
    Parameters:
        os - 数据结构
        k - 标号为k的数据的索引值
    Returns:
        None
    """
    def update_Ek(os, k):
        # 计算Ek
        Ek = cal_Ek(os, k)
        # 更新误差缓存
        os.ecache[k] = [1, Ek]
```

接下来将简单介绍一下用于寻找决策边界的优化例程:

```
    """
    函数说明: 优化的SMO算法
    Parameters:
        i - 标号为i的数据的索引值
        os - 数据结构
    Returns:
        1 - 有任意一对alpha值发生变化
        0 - 没有任意一对alpha值发生变化或变化太小
    """
    def innerL(i, os):
        # 步骤1: 计算误差Ei
        Ei = cal_Ek(os, i)
        # 优化alpha,设定一定的容错率
        if((os.label_mat[i] * Ei < -os.tol) and (os.alphas[i] < os.C)) or ((os.
                                                label_mat[i] * Ei > os.tol) and (
                                                os.alphas[i] > 0)):
            # 使用内循环启发方式2选择alpha_j,并计算Ej
            j, Ej = select_j(i, os, Ei) #这里不再是随机选取了
            # 保存更新前的alpha值, 使用深层拷贝
            alpha_i_old = os.alphas[i].copy()
            alpha_j_old = os.alphas[j].copy()
            # 步骤2: 计算上界H和下界L
            if(os.label_mat[i] != os.label_mat[j]):
                L = max(0, os.alphas[j] - os.alphas[i])
                H = min(os.C, os.C + os.alphas[j] - os.alphas[i])
            else:
```

```
        L = max(0, os.alphas[j] + os.alphas[i] - os.C)
        H = min(os.C, os.alphas[j] + os.alphas[i])
    if L == H:
        print("L == H")
        return 0
    # 步骤3: 计算eta
    eta = 2.0 * os.X[i, :] * os.X[j, :].T - os.X[i, :] * os.X[i, :].T -
                                            os.X[j, :] * os.X[j, :].T
    if eta >= 0:
        print("eta >= 0")
        return 0
    # 步骤4: 更新alpha_j
    os.alphas[j] -= os.label_mat[j] * (Ei - Ej) / eta
    # 步骤5: 修剪alpha_j
    os.alphas[j] = clip_alpha(os.alphas[j], H, L)
    # 更新Ej至误差缓存
    update_Ek(os, j)
    if(abs(os.alphas[j] - alpha_j_old) < 0.00001):
        print("alpha_j变化太小")
        return 0
    # 步骤6: 更新alpha_i
    os.alphas[i] += os.label_mat[i] * os.label_mat[j] * (alpha_j_old -
                                            os.alphas[j])
    # 更新Ei至误差缓存
    update_Ek(os, i)
    # 步骤7: 更新b_1和b_2:
    b1 = os.b - Ei - os.label_mat[i] * (os.alphas[i] - alpha_i_old) *
                                            os.X[i, :] * os.X[i, :].T -
                                            os.label_mat[j] * (os.alphas[
                                            j] - alpha_j_old) * os.X[j, :
                                            ] * os.X[i, :].T
    b2 = os.b - Ej - os.label_mat[i] * (os.alphas[i] - alpha_i_old) *
                                            os.X[i, :] * os.X[j, :].T -
                                            os.label_mat[j] * (os.alphas[
                                            j] - alpha_j_old) * os.X[j, :
                                            ] * os.X[j, :].T
    # 步骤8: 根据b_1和b_2更新b
    if(0 < os.alphas[i] < os.C):
        os.b = b1
```

```
        elif(0 < os.alphas[j] < os.C):
            os.b = b2
        else:
            os.b = (b1 + b2) / 2.0
        return 1 #表示有更新
    else:
        return 0 #表示没有更新
```

代码几乎与上一节中给出的 smo-simple() 函数一模一样，但是这里的代码已经使用了自己的数据结构。该结构在参数 os 中传递。

第二个重要的修改就是使用程序 select-j() 而不是 select-j-random() 来选择第二个 alpha 的值。最后，在 alpha 值改变时更新 ecache。

完整版 Platt SMO 的外循环代码:

```
"""
函数说明: 完整的线性SMO算法
Parameters:
    dataMatIn - 数据矩阵
    classLabels - 数据标签
    C - 松弛变量
    toler - 容错率
    maxIter - 最大迭代次数
Returns:
    oS.b - SMO算法计算的b
    oS.alphas - SMO算法计算的alphas
"""
def smo_p(data_mat_in, class_labels, C, toler, max_iter):
    # 初始化数据结构
    os = opt_struct(np.mat(data_mat_in), np.mat(class_labels).transpose(),
                                            C, toler)
    # 初始化当前迭代次数
    iter = 0
    entrie_set = True #是否在全部数据集上迭代
    alpha_pairs_changed = 0
    # 遍历整个数据集alpha都没有更新或者超过最大迭代次数, 则退出循环
    while(iter < max_iter) and ((alpha_pairs_changed > 0) or (entrie_set)):
        alpha_pairs_changed = 0
        if entrie_set: # 遍历整个数据集
            for i in range(os.m):
```

```
                # 使用优化的SMO算法
                alpha_pairs_changed += innerL(i, os) #innerL返回的值是0或者
                                                     1
                print("全样本遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (
                                                     iter, i,
                                                     alpha_pairs_changed))
            iter += 1
        # 遍历非边界值
        else:
            # 遍历不在边界0和C的alpha
            non_bound_i_s = np.nonzero((os.alphas.A > 0) * (os.alphas.A < C
                                                    ))[0]
            for i in non_bound_i_s:
                alpha_pairs_changed += innerL(i, os)
                print("非边界遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (
                                                     iter, i,
                                                     alpha_pairs_changed))
            iter += 1
        # 遍历一次后改为非边界遍历
        if entrie_set:
            entrie_set = False #进行切换，遍历非边界数据集
        # 如果alpha没有更新，计算全样本遍历
        elif(alpha_pairs_changed == 0):
            entrie_set = True
        print("迭代次数:%d" % iter)
    # 返回SMO算法计算的b和alphas
    return os.b, os.alphas
```

其输入和函数 smo-simple() 完全一样。函数一开始构建一个数据结构来容纳所有的数据，然后需要对控制函数退出的一些变量进行初始化。整个代码的主体是 while 循环，这与 smo-simple() 有些类似，但是这里的循环退出条件更多一些。

当迭代次数超过指定的最大值，或者遍历整个集合都未对任意 alpha 对进行修改时，就退出循环。

这里的 max-iter 变量和函数 smo-simple() 中的作用有一点不同，后者当没有任何 alpha 发生改变时会将整个集合的一次遍历过程计成一次迭代，而这里的一次迭代定义为一次循环过程，而不管该循环具体做了什么事。此时，如果在优化过程中存在波动就会停止，因此这里的做法优于 smo-simple() 函数中的计数方法。

while 循环的内部与 smo-simple() 中有所不同，一开始的 for 循环在数据集上遍历任意可能的 alpha 。

我们通过调用 innerL() 来选择第二个 alpha，并在可能时对其进行优化处理。如果有任意一对 alpha 值发生改变，那么会返回 1。第二个 for 循环遍历所有的非边界 alpha 值，也就是不在边界 0 或 C 上的值。

接下来，我们对 for 循环在非边界循环和完整遍历之间进行切换，并打印出迭代次数。最后程序将会返回常数 b 和 alpha 值。

我们的结果如下:

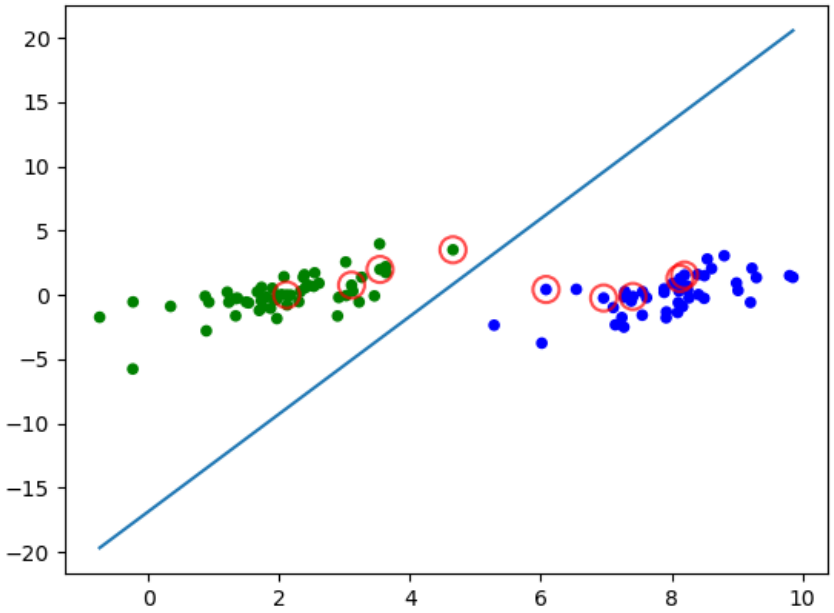
```
L == H
全样本遍历:第0次迭代 样本:0, alpha优化次数:0
L == H
全样本遍历:第0次迭代 样本:1, alpha优化次数:0
全样本遍历:第0次迭代 样本:2, alpha优化次数:1
L == H
全样本遍历:第0次迭代 样本:3, alpha优化次数:1
全样本遍历:第0次迭代 样本:4, alpha优化次数:2
全样本遍历:第0次迭代 样本:5, alpha优化次数:2
全样本遍历:第0次迭代 样本:6, alpha优化次数:2
alpha_j变化太小
......
全样本遍历:第2次迭代 样本:96, alpha优化次数:0
alpha_j变化太小
全样本遍历:第2次迭代 样本:97, alpha优化次数:0
全样本遍历:第2次迭代 样本:98, alpha优化次数:0
全样本遍历:第2次迭代 样本:99, alpha优化次数:0
迭代次数:3
```

我们生成的参数值为:

```
data_array, class_labels = load_dataset('testSet.txt')
b, alphas = smo_p(data_array, class_labels, 0.6, 0.001, 40)
w = cal_w_s(alphas, data_array, class_labels)
print('b=', b)
print('w=', w)
print('alphas=', alphas[alphas > 0])
for i in range(100):
  if alphas[i] > 0.0:
     print(data_array[i], class_labels[i])
```

```
b= [[-2.89901748]]
w= [[ 0.65307162]
 [-0.17196128]]
alphas= [[0.06961952 0.0169055  0.0169055  0.0272699  0.04522972 0.0272699
  0.0243898  0.06140181 0.06140181]]
[3.542485, 1.977398] -1.0
[2.114999, -0.004466] -1.0
[8.127113, 1.274372] 1.0
[4.658191, 3.507396] -1.0
[8.197181, 1.545132] 1.0
[7.40786, -0.121961] 1.0
[6.960661, -0.245353] 1.0
[6.080573, 0.418886] 1.0
[3.107511, 0.758367] -1.0
```

我们最终的决策边界和支持向量可视化为:



3. 在复杂数据上应用核函数

我们上面的 SMO 算法核函数其实就是线性可分的，那么对于非线性可分的呢？接下来，我们就要使用一种称为核函数的工具将数据转换成易分类器理解的形式。

径向基核函数

径向基函数是 SVM 中常用的一个核函数。

径向基函数是一个采用向量作为自变量的函数，能够基于向量距离运算输出一个标量。这个距离可以是从 <0,0> 向量或者其他向量开始计算的距离。

接下来，我们将会使用到径向基函数的高斯版本，其具体公式为：

$$K(x,y) = exp(\frac{-||x-y||^2}{2\sigma^2})$$

其中，$sigma$ 是用户定义的用于确定到达率或者说函数值跌落到 0 的速度参数。

上述高斯核函数将数据从其特征空间映射到更高维的空间，具体来说这里是映射到一个无穷维的空间。

高斯核函数只是一个常用的核函数，使用者并不需要确切地理解数据到底是如何表现的，而且使用高斯核函数还会得到一个理想的结果。

我们定义核函数如下:

```
"""
函数说明：通过核函数将数据转换更高维空间
Parameters:
    X - 数据矩阵
    A - 单个数据的向量
    kTup - 包含核函数信息的元组
Returns:
    K - 计算的核K
"""
def kernel_trans(xi, xj, kTup):
    """
    :param kTup: 两维，第一列是字符串，为lin,rbf，如果是rbf，第二列多一个
                                                    sigmma
    :return:
    """
    # 读取X的行列数
    m, n = np.shape(xi)
    # K初始化为m行1列的零向量
    K = np.mat(np.zeros((m, 1)))
    # 线性核函数只进行内积
    if kTup[0] == 'lin':
        K = xi * xj.T
    # 高斯核函数，根据高斯核函数公式计算
```

```
    elif kTup[0] == 'rbf':
        for j in range(m):
            delta_row = xi[j, :] - xj
            K[j] = delta_row * delta_row.T
        K = np.exp(K / (-1 * kTup[1] ** 2))
    else:
        raise NameError('核函数无法识别')
    return K
```

此时我们定义的数据结构为:

```
"""
类说明: 维护所有需要操作的值
Parameters:
    data_mat_in - 数据矩阵
    class_cabels - 数据标签
    C - 松弛变量
    toler - 容错率
Returns:
    None
"""
"""
定义一个新的数据结构
"""
class opt_struct:
    def __init__(self, data_mat_in, class_labels, C, toler,kTup):
        # 数据矩阵
        self.X = data_mat_in #传进来的数据
        # 数据标签
        self.label_mat = class_labels
        # 松弛变量
        self.C = C
        # 容错率
        self.tol = toler
        # 矩阵的行数
        self.m = np.shape(data_mat_in)[0]
        # 根据矩阵行数初始化alphas矩阵, 一个m行1列的全零列向量
        self.alphas = np.mat(np.zeros((self.m, 1)))
        # 初始化b参数为0
        self.b = 0
```

```
        # 根据矩阵行数初始化误差缓存矩阵，第一列为是否有效标志位，其中0无
                                                效，1有效;第二列为实际的误差
                                                Ei的值
        """
        我们之前的定义为:Ei=gxi-yi
        yi是标签的实际值。
        gx=alpha_i*y_i*x_i.x,就相当于是w.x+b
        因为误差值经常用到，所以希望每次计算后放到一个缓存当中，将ecache一
                                                分为二，第一列是标志位，取值
                                                为0或者1，为1时表示已经算出来
        """
        self.ecache = np.mat(np.zeros((self.m, 2)))
        self.K=np.mat(np.zeros((self.m,self.m)))
        for i in range(self.m):
            self.K[:,i]=kernel_trans(self.X,self.X[i,:],kTup)
```

引入了一个新变量 kTup，kTup 是一个包含核函数信息的元组，元组的第一个参数是描述所用核函数类型的一个字符串，其他 2 个参数则都是核函数可能需要的可选参数。该函数首先构建出了一个列向量，然后检查元组以确定核函数的类型。这里只给出了 2 种选择，但是依然可以很容易地通过添加 elif 语句来扩展到更多选项。

在线性核函数的情况下，内积计算在“所有数据集”和“数据集中的一行”这两个输入之间展开。在径向基核函数的情况下，在 for 循环中对于矩阵的每个元素计算高斯函数的值。而在 for 循环结束之后，我们将计算过程应用到整个向量上去。值得一提的是，在 NumPy 矩阵中，除法符号意味着对矩阵元素展开计算。

为了使用核函数，先期的两个函数 innerL() 和 cal-Ek() 的代码需要做些修改。

```
"""
函数说明：优化的SMO算法
Parameters:
    i - 标号为i的数据的索引值
    oS - 数据结构
Returns:
    1 - 有任意一对alpha值发生变化
    0 - 没有任意一对alpha值发生变化或变化太小
"""
def innerL(i, os):
    # 步骤1：计算误差Ei
    Ei = cal_Ek(os, i)
```

```
    # 优化alpha,设定一定的容错率
    if((os.label_mat[i] * Ei < -os.tol) and (os.alphas[i] < os.C)) or ((os.
                                        label_mat[i] * Ei > os.tol) and (
                                        os.alphas[i] > 0)):
        # 使用内循环启发方式2选择alpha_j,并计算Ej
        j, Ej = select_j(i, os, Ei)
        # 保存更新前的alpha值, 使用深层拷贝
        alpha_i_old = os.alphas[i].copy()
        alpha_j_old = os.alphas[j].copy()
        # 步骤2: 计算上界H和下界L
        if(os.label_mat[i] != os.label_mat[j]):
            L = max(0, os.alphas[j] - os.alphas[i])
            H = min(os.C, os.C + os.alphas[j] - os.alphas[i])
        else:
            L = max(0, os.alphas[j] + os.alphas[i] - os.C)
            H = min(os.C, os.alphas[j] + os.alphas[i])
        if L == H:
            print("L == H")
            return 0
        # 步骤3: 计算eta
        eta = 2.0 * os.K[i, j] - os.K[i, i] - os.K[j, j] #这里的计算就要采
                                                用核函数了
        if eta >= 0:
            print("eta >= 0")
            return 0
        # 步骤4: 更新alpha_j
        os.alphas[j] -= os.label_mat[j] * (Ei - Ej) / eta
        # 步骤5: 修剪alpha_j
        os.alphas[j] = clip_alpha(os.alphas[j], H, L)
        # 更新Ej至误差缓存
        update_Ek(os, j)
        if(abs(os.alphas[j] - alpha_j_old) < 0.00001):
            print("alpha_j变化太小")
            return 0
        # 步骤6: 更新alpha_i
        os.alphas[i] += os.label_mat[i] * os.label_mat[j] * (alpha_j_old -
                                                os.alphas[j])
        # 更新Ei至误差缓存
        update_Ek(os, i)
```

```
        # 步骤7: 更新b_1和b_2:
        b1 = os.b - Ei - os.label_mat[i] * (os.alphas[i] - alpha_i_old) *
                                            os.K[i, i] - os.label_mat[j]
                                            * (os.alphas[j] - alpha_j_old
                                            ) * os.K[j, i]
        b2 = os.b - Ej - os.label_mat[i] * (os.alphas[i] - alpha_i_old) *
                                            os.K[i, j] - os.label_mat[j]
                                            * (os.alphas[j] - alpha_j_old
                                            ) * os.K[j, j]
        # 步骤8: 根据b_1和b_2更新b
        if(0 < os.alphas[i] < os.C):
            os.b = b1
        elif(0 < os.alphas[j] < os.C):
            os.b = b2
        else:
            os.b = (b1 + b2) / 2.0
        return 1
    else:
        return 0
```

```
"""
函数说明: 计算误差
Parameters:
    os - 数据结构
    k - 标号为k的数据
Returns:
    Ek - 标号为k的数据误差
"""
def cal_Ek(os, k):
    # multiply(a,b)就是个乘法, 如果a,b是两个数组, 那么对应元素相乘
    # .T为转置
    fXk = float(np.multiply(os.alphas, os.label_mat).T * os.K[:, k]+os.b)
    # 计算误差项
    Ek = fXk - float(os.label_mat[k])
    # 返回误差项
    return Ek
```

3. 在测试中使用核函数

接下来我们将构建一个对上节中的数据点进行有效分类的分类器，该分类器使用了径向基核函数。前面提到的径向基函数有一个用户定义的输入。首先，我们需要确定它的大小，然后利用该核函数构建出一个分类器。

利用核函数进行分类的径向基测试函数如下所示:

```
"""
函数说明：测试函数
Parameters:
    k1 - 使用高斯核函数的时候表示到达率
Returns:
    None
"""
def test_rbf(k1 = 1.3):
    # 加载训练集
    data_array, label_array = load_dataset('testSetRBF.txt')
    # 根据训练集计算b, alphas
    b, alphas = smo_p(data_array, label_array, 200, 0.0001, 100, ('rbf', k1
                                                                  ))
    data_mat = np.mat(data_array)
    label_mat = np.mat(label_array).transpose()
    # 获得支持向量
    svInd = np.nonzero(alphas.A > 0)[0]
    sVs = data_mat[svInd]
    labelSV = label_mat[svInd]
    print("支持向量个数:%d" % np.shape(sVs)[0])
    m, n = np.shape(data_mat)
    error_count = 0
    for i in range(m):
        # 计算各个点的核
        kernel_eval = kernel_trans(sVs, data_mat[i, :], ('rbf', k1))
        # 根据支持向量的点计算超平面，返回预测结果
        predict = kernel_eval.T * np.multiply(labelSV, alphas[svInd]) + b
        # 返回数组中各元素的正负号，用1和-1表示，并统计错误个数
        if np.sign(predict) != np.sign(label_array[i]):
            error_count += 1
    # 打印错误率
    print('训练集错误率:%.2f%%' % ((float(error_count) / m) * 100))
```

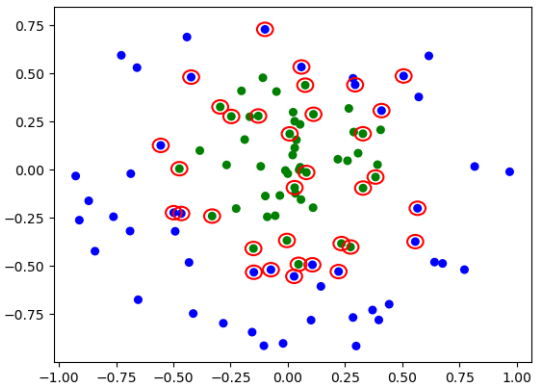
```
    # 加载测试集
    data_array, label_array = load_dataset('testSetRBF2.txt')
    error_count = 0
    data_mat = np.mat(data_array)
    label_mat = np.mat(label_array).transpose()
    m, n = np.shape(data_mat)
    for i in range(m):
        # 计算各个点的核
        kernel_eval = kernel_trans(sVs, data_mat[i, :], ('rbf', k1))
        # 根据支持向量的点计算超平面，返回预测结果
        predict = kernel_eval.T * np.multiply(labelSV, alphas[svInd]) + b
        # 返回数组中各元素的正负号，用1和-1表示，并统计错误个数
        if np.sign(predict) != np.sign(label_array[i]):
            error_count += 1
    # 打印错误率
    print('测试集错误率:%.2f%%' % ((float(error_count) / m) * 100))
```

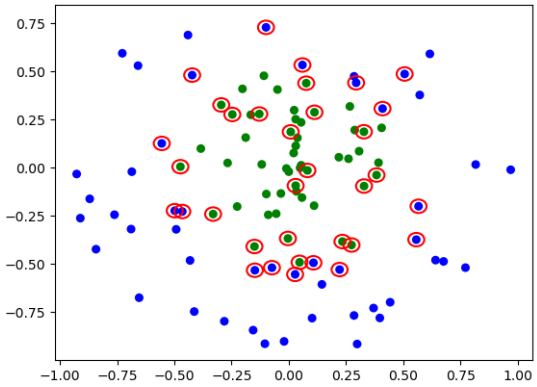
上述代码只有一个可选的输入参数，该输入参数是高斯径向基函数中的一个用户定义变量。

整个代码中最重要的是 for 循环开始的那两行，它们给出了如何利用核函数进行分类。首先利用结构初始化方法中使用过的 kernel-trans() 函数，得到转换后的数据。然后，再用其与前面的 alpha 及类别标签值求积。其中需要特别注意的另一件事是，在这几行代码中，是如何做到只需要支持向量数据就可以进行分类的。除此之外，其他数据都可以直接舍弃。

与第一个 for 循环相比，第二个 for 循环仅仅只有数据集不同，后者采用的是测试数据集。

我们分类后的支持向量可视化为:





其中，第一张图是关于训练集的，第二张图是关于测试集的。

我们可以尝试更换不同的 k1 参数以观察测试错误率、训练错误率、支持向量个数随 k1 的变化情况。

这次不再赘述。

我们看一下在训练集和测试集上的误差率:

```
支持向量个数:30
训练集错误率:13.00%
测试集错误率:15.00%
```

支持向量的数目存在一个最优值。SVM 的优点在于它能对数据进行高效分类。如果支持向量太少，就可能会得到一个很差的决策边界（下个例子会说明这一点）；如果支持向量太多，也就相当于每次都利用整个数据集进行分类，这种分类方法称为 k 近邻。

我们回到 K 近邻方法中的那个例子，手写识别问题回顾。

4. 示例：手写识别问题回顾

基于 SVM 的手写数字识别

代码如下:

```
"""
函数说明：加载图片
Parameters:
    dirName - 文件夹名字
Returns:
    trainingMat - 数据矩阵
    data_labels - 数据标签
```

```
"""
def load_images(dir_name):
    from os import listdir
    # 测试集的Labels
    data_labels = []
    # 返回trainingDigits目录下的文件名
    training_file_list = listdir(dir_name)
    # 返回文件夹下文件的个数
    m = len(training_file_list)
    # 初始化训练的Mat矩阵（全零阵），测试集
    training_mat = np.zeros((m, 1024))
    # 从文件名中解析出训练集的类别
    for i in range(m):
        # 获得文件的名字
        file_name_string = training_file_list[i]
        file_string = file_name_string.split('.')[0]
        # 获得分类的数字
        class_number = int(file_string.split('_')[0])
        if class_number == 9:
            data_labels.append(-1)
        else:
            data_labels.append(1)
        training_mat[i, :] = imgage_vector('%s/%s' % (dir_name,
                                                      file_name_string))
    return training_mat, data_labels

"""
函数说明：测试函数
Parameters:
    kTup - 包含核函数信息的元组
Returns:
    None
"""
def test_digits(kTup=('rbf', 10)):
    # 加载训练集
    data_array, label_array = load_images('trainingDigits')
    # 根据训练集计算b, alphas
    b, alphas = smo_p(data_array, label_array, 200, 0.001, 10, kTup)
    data_mat = np.mat(data_array)
```

```
    label_mat = np.mat(label_array).transpose()
    # 获得支持向量
    svInd = np.nonzero(alphas.A > 0)[0]
    sVs = data_mat[svInd]
    labelSV = label_mat[svInd]
    print("支持向量个数:%d" % np.shape(sVs)[0])
    m, n = np.shape(data_mat)
    error_count = 0
    for i in range(m):
        # 计算各个点的核
        kernel_eval = kernel_trans(sVs, data_mat[i, :], kTup)
        # 根据支持向量的点计算超平面，返回预测结果
        predict = kernel_eval.T * np.multiply(labelSV, alphas[svInd]) + b
        # 返回数组中各元素的正负号，用1和-1表示，并统计错误个数
        if np.sign(predict) != np.sign(label_mat[i]):
            error_count += 1
    # 打印错误率
    print('训练集错误率:%.2f%%' % (float(error_count) / m))
    # 加载测试集
    data_array, label_array = load_images('testDigits')
    error_count = 0
    data_mat = np.mat(data_array)
    label_mat = np.mat(label_array).transpose()
    m, n = np.shape(data_mat)
    for i in range(m):
        # 计算各个点的核
        kernel_eval = kernel_trans(sVs, data_mat[i, :], kTup)
        # 根据支持向量的点计算超平面，返回预测结果
        predict = kernel_eval.T * np.multiply(labelSV, alphas[svInd]) + b
        # 返回数组中各元素的正负号，用1和-1表示，并统计错误个数
        if np.sign(predict) != np.sign(label_array[i]):
            error_count += 1
    # 打印错误率
    print('测试集错误率:%.2f%%' % (float(error_count) / m))
```

```
def hand_writing_class_test():
    import os
    data_labels = []
    # 导入训练集，‘trainingDigits’是一个文件夹
```

```
    training_file_list = os.listdir('trainingDigits')
    # 计算训练样本个数
    m = len(training_file_list)
    # 初始化数据集，将所有训练数据用一个m行，1024列的矩阵表示
    training_mat = np.zeros((m,1024))
    for i in range(m):
        # 获得所有文件名，文件名格式‘x_y.txt’,x表示这个手写数字实际表示的
                                                数字（label）
        file_name_string = training_file_list[i]
        # 去除 .txt
        file_str = file_name_string.split('.')[0]
        # classnumber为每个样本的分类，用‘_’分割，取得label
        class_num_str = int(file_str.split('_')[0])
        # 将所有标签都存进hwLables[]
        data_labels.append(class_num_str)
        # 将文件转化为向量后存入trainingMat[],这里展现了灵活的文件操作
        training_mat[i,:] = imgage_vector('./trainingDigits/%s' %
                                                file_name_string)
    test_file_list = os.listdir('testDigits')  #迭代测试集
    error_count = 0.0
    m_test = len(test_file_list)
    for i in range(m_test):
        file_name_string = test_file_list[i]
        # 去除 .txt
        file_str = file_name_string.split('.')[0]
        class_num_string = int(file_str.split('_')[0])
        # 这部分针对测试集的预处理和前面基本相同
        vector_under_test = imgage_vector('./testDigits/%s' %
                                                file_name_string)
        # 使用算法预测样本所属类别，调用了前面写的classify0()函数
        classify_result = classify(vector_under_test, training_mat,
                                                data_labels, 3)
        # print ("分类结果:%d,真实类别:%d" % (classify_result,
                                                class_num_string))
        # 算法结果与样本的实际分类做对比
        if (classify_result != class_num_string): error_count += 1.0
    print('KNN算法的预测情况:')
    print ("\n分类错误的个数为: %d" % error_count)
    print ("\n分类错误率为: %f" % (error_count/float(m_test)))
```

函数 load-images() 是作为前面 k 近邻算法中的 hand-writing-class-test() 的一部分出现的。

它已经被重构为自身的一个函数。其中仅有的一个大区别在于，在 K 近邻中代码直接应用类别标签，而同支持向量机一起使用时，类别标签为-1 或者 +1。因此，一旦碰到数字 9，则输出类别标签 1，否则输出 +1。

本质上，支持向量机是一个二类分类器，其分类结果不是 +1 就是-1。由于这里我们只做二类分类，因此除了 1 和 9 之外的数字都被去掉了。

下一个函数 test-digits() 并不是全新的函数，它和 test-Rbf() 的代码几乎一样，唯一的大区别就是它调用了 load-images() 函数来获得类别标签和数据。另一个细小的不同是现在这里的函数元组 kTup 是输入参数，而在 test-Rbf() 中默认的就是使用 rbf 核函数。如果对于函数 test-digits() 不增加任何输入参数的话，那么 kTup 的默认值就是 ('rbf' ,10)。

我们的运行结果如下所示:

```
迭代次数:5
支持向量个数:311
训练集错误率:0.00%
测试集错误率:0.01%
KNN算法的预测情况:

分类错误的个数为: 10
分类错误率为: 0.010571
```

支持向量机试图通过求解一个二次优化问题来最大化分类间隔。在过去，训练支持向量机常采用非常复杂并且低效的二次规划求解方法。John Platt 引入了 SMO 算法，此算法可以通过每次只优化 2 个 alpha 值来加快 SVM 的训练速度。

本章首先讨论了一个简化版本所实现的 SMO 优化过程，接着给出了完整的 Platt SMO 算法。相对于简化版而言，完整版算法不仅大大地提高了优化的速度，还使其存在一些进一步提高运行速度的空间。

核方法或者说核技巧会将数据（有时是非线性数据）从一个低维空间映射到一个高维空间，可以将一个在低维空间中的非线性问题转换成高维空间下的线性问题来求解。核方法不止在 SVM 中适用，还可以用于其他算法中。而其中的径向基函数是一个常用的度量两个向量距离的核函数。

支持向量机是一个二类分类器。当用其解决多类问题时，则需要额外的方法对其进行扩展。SVM 的效果也对优化参数和所用核函数中的参数敏感。

第三章　源代码

1. 应用简化版 SMO 算法处理小规模数据集

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jul 18 2022
@author: wzk
"""
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
import random

"""
函数说明: 读取数据,处理文本数据
Parameters:
    file_name - 文件名
Returns:
    data_mat - 数据矩阵
    label_mat - 数据标签
"""
def load_dataset(file_name):
    # 数据矩阵
    data_mat = []
    # 标签向量
    label_mat = []
    # 打开文件
    fr = open(file_name)
    # 逐行读取
    for line in fr.readlines():
        # 去掉每一行首尾的空白符, 例如'\n','\r','\t',' '
```

```
        # 将每一行内容根据'\t'符进行切片
        line_array = line.strip().split('\t')
        # 添加数据(100个元素排成一行)
        data_mat.append([float(line_array[0]), float(line_array[1])])
        # 添加标签(100个元素排成一行)
        label_mat.append(int(line_array[2]))
    return data_mat, label_mat


"""
函数说明: 随机选择alpha_j。alpha的选取, 随机选择一个不等于i值的j
Parameters:
    i - 第一个alpha的下标
    m - alpha参数个数
Returns:
    j - 返回选定的数字
"""
def select_J_rand(i, m):
    """
    :param i: 表示alpha_i
    :param m: 表示样本数
    :return:
    """
    j = i
    while(j == i):
        # 如果i和j相等, 那么就从样本中随机选取一个, 可以认为j就是选择的
                                                    alpha_2
        # uniform()方法将随机生成一个实数, 它在[x, y)范围内
        j = int(random.uniform(0, m))
    return j

"""
函数说明: 修剪alpha
Parameters:
    a_j - alpha值
    H - alpha上限
    L - alpha下限
Returns:
    a_j - alpha值
```

```
用于调整大于H或小于L的alpha值。
"""
def clip_alpha(a_j, H, L):
    if a_j > H:
        a_j = H
    if L > a_j:
        a_j = L
    return a_j

"""
函数说明：简化版SMO算法
Parameters:
    data_mat_in - 数据矩阵
    class_labels - 数据标签
    C - 松弛变量
    toler - 容错率
    max_iter - 最大迭代次数
Returns:
    None
"""
def smo_simple(data_mat_in, class_labels, C, toler, max_iter):
    """
    :param data_mat_in: 相当于我们的x
    :param class_labels: 相当于我们的y
    :param C: 惩罚因子
    :param toler:
    :param max_iter:
    :return:
    """
    # 转换为numpy的mat矩阵存储(100,2)
    data_matrix = np.mat(data_mat_in) #相当于x
    # 转换为numpy的mat矩阵存储并转置(100,1)
    label_mat = np.mat(class_labels).transpose() #相当于y
    # 初始化b参数，统计data_matrix的维度,m:行; n:列
    b = 0
    # 统计dataMatrix的维度,m:100行; n:2列
    m, n = np.shape(data_matrix)
    # 初始化alpha参数，设为0
    alphas = np.mat(np.zeros((m, 1)))
```

```
    # 初始化迭代次数
    iter_num = 0
    # 最多迭代maxIter次
    while(iter_num < max_iter):
        alpha_pairs_changed = 0
        for i in range(m):
            # 步骤1: 计算误差Ei
            # multiply(a,b)就是个乘法，如果a,b是两个数组，那么对应元素相乘
            # .T为转置，转置的目的是因为后面的核函数是一个向量。核函数我们
                                                    最原始的方式，可以直接使
                                                    用点乘就可以，即x_i.x
            fxi = float(np.multiply(alphas, label_mat).T * (data_matrix *
                                                    data_matrix[i, :].T)) + b
            # 误差项计算公式
            Ei = fxi - float(label_mat[i])
            # 优化alpha，设定一定的容错率
            if((label_mat[i] * Ei < -toler) and (alphas[i] < C)) or ((
                                                    label_mat[i] * Ei > toler
                                                    ) and (alphas[i] > 0)):
                # 随机选择另一个alpha_i成对比优化的alpha_j
                j = select_J_rand(i, m)
                # 步骤1，计算误差Ej
                fxj = float(np.multiply(alphas, label_mat).T * (data_matrix
                                                     * data_matrix[j, :].
                                                    T)) + b
                # 误差项计算公式
                Ej = fxj - float(label_mat[j])
                # 保存更新前的alpha值，使用深拷贝(完全拷贝)A深层拷贝为B，A
                                                    和B是两个独立的个体
                alpha_i_old = alphas[i].copy()
                alpha_j_old = alphas[j].copy()
                # 步骤2: 计算上下界H和L
                if(label_mat[i] != label_mat[j]):
                    L = max(0, alphas[j]-alphas[i])
                    H = min(C, C + alphas[j] - alphas[i])
                else:
                    L = max(0, alphas[j] + alphas[i] - C)
                    H = min(C, alphas[j] + alphas[i])
                if(L == H):
```

```
            print("L == H")
            continue
        # 步骤3: 计算eta, 转置表示相乘没有错误, 相当于求的-eta
        eta = 2.0 * data_matrix[i, :] * data_matrix[j, :].T -
                                                data_matrix[i, :] *
                                                data_matrix[i, :].T -
                                                 data_matrix[j, :] *
                                                data_matrix[j, :].T
        if eta >= 0:
            print("eta>=0")
            continue
        # 步骤4: 更新alpha_j
        alphas[j] -= label_mat[j] * (Ei - Ej) / eta
        # 步骤5: 修剪alpha_j
        alphas[j] = clip_alpha(alphas[j], H, L)
        if(abs(alphas[j] - alpha_j_old) < 0.00001):
            print("alpha_j变化太小")
            continue
        # 步骤6: 更新alpha_i
        alphas[i] += label_mat[j] * label_mat[i] * (alpha_j_old -
                                                alphas[j])
        # 步骤7: 更新b_1和b_2
        b1 = b - Ei - label_mat[i] * (alphas[i] - alpha_i_old) *
                                                data_matrix[i, :] *
                                                data_matrix[i, :].T -
                                                 label_mat[j] * (
                                                alphas[j] -
                                                alpha_j_old) *
                                                data_matrix[j, :] *
                                                data_matrix[i, :].T
        b2 = b - Ej - label_mat[i] * (alphas[i] - alpha_i_old) *
                                                data_matrix[i, :] *
                                                data_matrix[j, :].T -
                                                 label_mat[j] * (
                                                alphas[j] -
                                                alpha_j_old) *
                                                data_matrix[j, :] *
                                                data_matrix[j, :].T
        # 步骤8: 根据b_1和b_2更新b
```

```
                if(0 < alphas[i] < C):
                    b = b1
                elif(0 < alphas[j] < C):
                    b = b2
                else:
                    b = (b1 + b2) / 2.0
                # 统计优化次数
                alpha_pairs_changed += 1
                # 打印统计信息
                print("第%d次迭代 样本: %d, alpha优化次数: %d" % (iter_num
                                                        , i,
                                                        alpha_pairs_changed))
        # 更新迭代次数
        if(alpha_pairs_changed == 0):
            iter_num += 1
        else:
            iter_num = 0
        print("迭代次数: %d" % iter_num)
    return b, alphas

"""
函数说明: 计算w
Returns:
    data_mat - 数据矩阵
    label_mat - 数据标签
    alphas - alphas值
Returns:
    w - 直线法向量
"""
def get_w(data_mat_in, class_labels, alphas):
    """
    :param data_mat: x
    :param label_mat: y
    :param alphas:
    :return:
    """
    data_mat=np.mat(data_mat_in)
    label_mat=np.mat(class_labels).transpose()
    m,n=np.shape(data_mat)
```

```
    # 初始化w都为1
    w=np.zeros((n,1))
    # dot()函数是矩阵乘，而*则表示逐个元素相乘
    # w = sum(alpha_i * yi * xi)
    #循环计算
    for i in range(m):
        w+=np.multiply(alphas[i]*label_mat[i],data_mat[i,:].T)
    return w

"""
函数说明：分类结果可视化
Returns:
    dataMat - 数据矩阵
    w - 直线法向量
    b - 直线截距
Returns:
    None
"""
def show_classifer(data_mat, w, b):
    data_mat, label_mat = load_dataset('testSet.txt')
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cm = mpl.colors.ListedColormap(['g', 'r'])
    ax.scatter(np.array(data_mat)[:, 0], np.array(data_mat)[:, 1], c=np.
                                            array(label_mat), cmap=cm, s=20)
    # 绘制直线
    x1 = max(data_mat)[0]
    x2 = min(data_mat)[0]
    a1, a2 = w
    b = float(b)
    a1 = float(a1[0])
    a2 = float(a2[0])
    y1, y2 = (-b - a1 * x1) / a2, (-b - a1 * x2) / a2
    plt.plot([x1, x2], [y1, y2])
    # 找出支持向量点
    # enumerate在字典上是枚举、列举的意思
    for i, alpha in enumerate(alphas):
        # 支持向量机的点
        if(abs(alpha) > 0):
```

```
            x, y = data_mat[i]
            plt.scatter([x], [y], s=150, c='none', alpha=0.7, linewidth=1.5
                                                    , edgecolors='red')
    plt.show()

if __name__ == '__main__':
    data_mat, label_mat = load_dataset('testSet.txt')
    print('数据标签为:\n',label_mat)
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cm = mpl.colors.ListedColormap(['g', 'r'])
    ax.scatter(np.array(data_mat)[:, 0], np.array(data_mat)[:, 1], c=np.
                                          array(label_mat), cmap=cm, s=20)

    b, alphas = smo_simple(data_mat, label_mat, 0.6, 0.001, 40)
    w = get_w(data_mat, label_mat, alphas)
    print('b=',b)
    print('w=',w)
    print('alphas=',alphas[alphas>0])
    for i in range(100):
        if alphas[i]>0.0:
            print(data_mat[i],label_mat[i])
    x = np.arange(-2.0, 12, 0.1)
    # w0x0+w1x1=0,x1=-w0x0/w1
    y = (-w[0] * x - b) / w[1]
    ax.plot(x,y.reshape(-1,1))
    ax.axis([-2,12,-8.6,7])
    plt.show()
    show_classifer(data_mat,w,b)
```

2. 利用完整 SMO 算法加速优化

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
import numpy as np
import random
import matplotlib as mpl
"""
```

```
改进SMO算法以加快我们的SVM运行速度!
"""


"""
函数说明: 读取数据
Parameters:
    file_name - 文件名
Returns:
    data_mat - 数据矩阵
    label_mat - 数据标签
"""
def load_dataset(file_name):
    # 数据矩阵
    data_mat = []
    # 标签向量
    label_mat = []
    # 打开文件
    fr = open(file_name)
    # 逐行读取
    for line in fr.readlines():
        # 去掉每一行首尾的空白符, 例如'\n','\r','\t',' '
        # 将每一行内容根据'\t'符进行切片
        line_array = line.strip().split('\t')
        # 添加数据(100个元素排成一行)
        data_mat.append([float(line_array[0]), float(line_array[1])])
        # 添加标签(100个元素排成一行)
        label_mat.append(float(line_array[2]))
    return data_mat, label_mat


"""
函数说明: 随机选择alpha_j
Parameters:
    i - alpha_i的索引值
    m - alpha参数个数
Returns:
    j - alpha_j的索引值
"""
def select_j_random(i, m):
    j = i
```

```
    while(j == i):
        # uniform()方法将随机生成一个实数，它在[x, y)范围内
        j = int(random.uniform(0, m))
    return j

"""
函数说明：修剪alpha_j
Parameters:
    aj - alpha_j值
    H - alpha上限
    L - alpha下限
Returns:
    aj - alpha_j值
"""
def clip_alpha(aj, H, L):
    if aj > H:
        aj = H
    if L > aj:
        aj = L
    return aj

"""
类说明：维护所有需要操作的值
Parameters:
    dataMatIn - 数据矩阵
    classLabels - 数据标签
    C - 松弛变量
    toler - 容错率
Returns:
    None
"""
"""
定义一个新的数据结构
"""
class opt_struct:
    def __init__(self, data_mat_in, class_labels, C, toler):
        # 数据矩阵
        self.X = data_mat_in #传进来的数据
        # 数据标签
```

```
        self.label_mat = class_labels
        # 松弛变量
        self.C = C
        # 容错率
        self.tol = toler
        # 矩阵的行数
        self.m = np.shape(data_mat_in)[0]
        # 根据矩阵行数初始化alphas矩阵，一个m行1列的全零列向量
        self.alphas = np.mat(np.zeros((self.m, 1)))
        # 初始化b参数为0
        self.b = 0
        # 根据矩阵行数初始化误差缓存矩阵，第一列为是否有效标志位，其中0无
                                                效，1有效;第二列为实际的误差
                                                Ei的值
        """
        我们之前的定义为:Ei=gxi-yi
        yi是标签的实际值。
        gx=alpha_i*y_i*x_i.x,就相当于是w.x+b
        因为误差值经常用到，所以希望每次计算后放到一个缓存当中，将ecache一
                                                分为二，第一列是标志位，取值
                                                为0或者1，为1时表示已经算出来
        """
        self.ecache = np.mat(np.zeros((self.m, 2)))


"""
函数说明：计算误差
Parameters:
    os - 数据结构
    k - 标号为k的数据
Returns:
    Ek - 标号为k的数据误差
"""
def cal_Ek(os, k):
    # multiply(a,b)就是个乘法，如果a,b是两个数组，那么对应元素相乘
    # .T为转置
    fXk = float(np.multiply(os.alphas, os.label_mat).T * (os.X * os.X[k, :]
                                            .T) + os.b)
    # 计算误差项
    Ek = fXk - float(os.label_mat[k])
```

```
    # 返回误差项
    return Ek


"""
函数说明：内循环启发方式2
选择第二个待优化的alpha_j，选择一个误差最大的alpha_j
即，我们在选择alpha_2的时候做了改进，选择误差最大的
Parameters:
    i - 标号为i的数据的索引值
    oS - 数据结构
    Ei - 标号为i的数据误差
Returns:
    j - 标号为j的数据的索引值
    maxK - 标号为maxK的数据的索引值
    Ej - 标号为j的数据误差
"""
def select_j(i, os, Ei):
    # 初始化
    max_K = -1 #下标的索引值
    max_delta_E = 0
    Ej = 0
    # 根据Ei更新误差缓存，即先计算alpha_1以及E1值
    os.ecache[i] = [1, Ei] #放入缓存当中，设为有效
    # 对一个矩阵.A转换为Array类型
    # 返回误差不为0的数据的索引值
    valid_ecache_list = np.nonzero(os.ecache[:, 0].A)[0] #找出缓存中不为0
    # 有不为0的误差
    if(len(valid_ecache_list) > 1):
        # 遍历，找到最大的Ek
        for k in valid_ecache_list: #迭代所有有效的缓存，找到误差最大的E
            # 不计算k==i节省时间
            if k == i: #不选择和i相等的值
                continue
            # 计算Ek
            Ek = cal_Ek(os, k)
            # 计算|Ei - Ek|
            delta_E = abs(Ei - Ek)
            # 找到maxDeltaE
```

```
            if(delta_E > max_delta_E):
                max_K = k
                max_delta_E = delta_E
                Ej = Ek
        # 返回max_K, Ej
        return max_K, Ej #这样我们就得到误差最大的索引值和误差最大的值
    # 没有不为0的误差
    else: #第一次循环时是没有有效的缓存值的, 所以随机选一个（仅会执行一次）
        # 随机选择alpha_j的索引值
        j = select_j_random(i, os.m)
        # 计算Ej
        Ej = cal_Ek(os, j)
    # 返回j, Ej
    return j, Ej

"""
函数说明: 计算Ek,并更新误差缓存
Parameters:
    os - 数据结构
    k - 标号为k的数据的索引值
Returns:
    None
"""
def update_Ek(os, k):
    # 计算Ek
    Ek = cal_Ek(os, k)
    # 更新误差缓存
    os.ecache[k] = [1, Ek]

"""
函数说明: 优化的SMO算法
Parameters:
    i - 标号为i的数据的索引值
    os - 数据结构
Returns:
    1 - 有任意一对alpha值发生变化
    0 - 没有任意一对alpha值发生变化或变化太小
"""
def innerL(i, os):
```

```
    # 步骤1: 计算误差Ei
    Ei = cal_Ek(os, i)
    # 优化alpha,设定一定的容错率
    if((os.label_mat[i] * Ei < -os.tol) and (os.alphas[i] < os.C)) or ((os.
                                            label_mat[i] * Ei > os.tol) and (
                                            os.alphas[i] > 0)):
        # 使用内循环启发方式2选择alpha_j,并计算Ej
        j, Ej = select_j(i, os, Ei) #这里不再是随机选取了
        # 保存更新前的alpha值, 使用深层拷贝
        alpha_i_old = os.alphas[i].copy()
        alpha_j_old = os.alphas[j].copy()
        # 步骤2: 计算上界H和下界L
        if(os.label_mat[i] != os.label_mat[j]):
            L = max(0, os.alphas[j] - os.alphas[i])
            H = min(os.C, os.C + os.alphas[j] - os.alphas[i])
        else:
            L = max(0, os.alphas[j] + os.alphas[i] - os.C)
            H = min(os.C, os.alphas[j] + os.alphas[i])
        if L == H:
            print("L == H")
            return 0
        # 步骤3: 计算eta
        eta = 2.0 * os.X[i, :] * os.X[j, :].T - os.X[i, :] * os.X[i, :].T -
                                                os.X[j, :] * os.X[j, :].T
        if eta >= 0:
            print("eta >= 0")
            return 0
        # 步骤4: 更新alpha_j
        os.alphas[j] -= os.label_mat[j] * (Ei - Ej) / eta
        # 步骤5: 修剪alpha_j
        os.alphas[j] = clip_alpha(os.alphas[j], H, L)
        # 更新Ej至误差缓存
        update_Ek(os, j)
        if(abs(os.alphas[j] - alpha_j_old) < 0.00001):
            print("alpha_j变化太小")
            return 0
        # 步骤6: 更新alpha_i
        os.alphas[i] += os.label_mat[i] * os.label_mat[j] * (alpha_j_old -
                                                os.alphas[j])
```

```
        # 更新Ei至误差缓存
        update_Ek(os, i)
        # 步骤7: 更新b_1和b_2:
        b1 = os.b - Ei - os.label_mat[i] * (os.alphas[i] - alpha_i_old) *
                                            os.X[i, :] * os.X[i, :].T -
                                            os.label_mat[j] * (os.alphas[
                                            j] - alpha_j_old) * os.X[j, :
                                            ] * os.X[i, :].T
        b2 = os.b - Ej - os.label_mat[i] * (os.alphas[i] - alpha_i_old) *
                                            os.X[i, :] * os.X[j, :].T -
                                            os.label_mat[j] * (os.alphas[
                                            j] - alpha_j_old) * os.X[j, :
                                            ] * os.X[j, :].T
        # 步骤8: 根据b_1和b_2更新b
        if(0 < os.alphas[i] < os.C):
            os.b = b1
        elif(0 < os.alphas[j] < os.C):
            os.b = b2
        else:
            os.b = (b1 + b2) / 2.0
        return 1 #表示有更新
    else:
        return 0 #表示没有更新

"""
函数说明: 完整的线性SMO算法
Parameters:
    dataMatIn - 数据矩阵
    classLabels - 数据标签
    C - 松弛变量
    toler - 容错率
    maxIter - 最大迭代次数
Returns:
    oS.b - SMO算法计算的b
    oS.alphas - SMO算法计算的alphas
"""
def smo_p(data_mat_in, class_labels, C, toler, max_iter):
    # 初始化数据结构
    os = opt_struct(np.mat(data_mat_in), np.mat(class_labels).transpose(),
```

```
                                        C, toler)
    # 初始化当前迭代次数
    iter = 0
    entrie_set = True #是否在全部数据集上迭代
    alpha_pairs_changed = 0
    # 遍历整个数据集alpha都没有更新或者超过最大迭代次数, 则退出循环
    while(iter < max_iter) and ((alpha_pairs_changed > 0) or (entrie_set)):
        alpha_pairs_changed = 0
        if entrie_set: # 遍历整个数据集
            for i in range(os.m):
                # 使用优化的SMO算法
                alpha_pairs_changed += innerL(i, os) #innerL返回的值是0或者
                                                     1
                print("全样本遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (
                                                     iter, i,
                                                     alpha_pairs_changed))
            iter += 1
        # 遍历非边界值
        else:
            # 遍历不在边界0和C的alpha
            non_bound_i_s = np.nonzero((os.alphas.A > 0) * (os.alphas.A < C
                                                    ))[0]
            for i in non_bound_i_s:
                alpha_pairs_changed += innerL(i, os)
                print("非边界遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (
                                                     iter, i,
                                                     alpha_pairs_changed))
            iter += 1
        # 遍历一次后改为非边界遍历
        if entrie_set:
            entrie_set = False #进行切换, 遍历非边界数据集
        # 如果alpha没有更新, 计算全样本遍历
        elif(alpha_pairs_changed == 0):
            entrie_set = True
        print("迭代次数:%d" % iter)
    # 返回SMO算法计算的b和alphas
    return os.b, os.alphas

"""
```

```
    函数说明：分类结果可视化
    Returns:
        dataMat - 数据矩阵
        classLabels - 数据标签
        w - 直线法向量
        b - 直线截距
    Returns:
        None
    """
    def show_classifer(data_mat, class_labels, w, b):
        data_mat, label_mat = load_dataset('testSet.txt')
        fig = plt.figure()
        ax = fig.add_subplot(111)
        cm = mpl.colors.ListedColormap(['g', 'b'])
        ax.scatter(np.array(data_mat)[:, 0], np.array(data_mat)[:, 1], c=np.
                                                array(label_mat), cmap=cm, s=20)
        # 绘制直线
        x1 = max(data_mat)[0]
        x2 = min(data_mat)[0]
        a1, a2 = w
        b = float(b)
        a1 = float(a1[0])
        a2 = float(a2[0])
        y1, y2 = (-b - a1 * x1) / a2, (-b - a1 * x2) / a2
        plt.plot([x1, x2], [y1, y2])
        # 找出支持向量点
        # enumerate在字典上是枚举、列举的意思
        for i, alpha in enumerate(alphas):
            # 支持向量机的点
            if (abs(alpha) > 0):
                x, y = data_mat[i]
                plt.scatter([x], [y], s=150, c='none', alpha=0.7, linewidth=1.5
                                                        , edgecolors='red')
        plt.show()
    """
    函数说明：计算w
    Returns:
        dataArr - 数据矩阵
        classLabels - 数据标签
```

```
    alphas - alphas值
Returns:
    w - 直线法向量
"""
def cal_w_s(alphas, data_array, class_labels):
    X = np.mat(data_array)
    label_mat = np.mat(class_labels).transpose()
    m, n = np.shape(X)
    w = np.zeros((n, 1))
    for i in range(m):
        w += np.multiply(alphas[i] * label_mat[i], X[i, :].T)
    return w


if __name__ == '__main__':
    data_array, class_labels = load_dataset('testSet.txt')
    b, alphas = smo_p(data_array, class_labels, 0.6, 0.001, 40)
    w = cal_w_s(alphas, data_array, class_labels)
    print('b=', b)
    print('w=', w)
    print('alphas=', alphas[alphas > 0])
    for i in range(100):
        if alphas[i] > 0.0:
            print(data_array[i], class_labels[i])
    show_classifer(data_array, class_labels, w, b)
```

3. 在复杂数据上应用核函数

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
import numpy as np
import random
import matplotlib as mpl
from matplotlib.patches import Circle
"""
函数说明：读取数据
Parameters:
    file_name - 文件名
```

```
    Returns:
        data_mat - 数据矩阵
        label_mat - 数据标签
    """
    def load_dataset(file_name):
        # 数据矩阵
        data_mat = []
        # 标签向量
        label_mat = []
        # 打开文件
        fr = open(file_name)
        # 逐行读取
        for line in fr.readlines():
            # 去掉每一行首尾的空白符，例如'\n','\r','\t',' '
            # 将每一行内容根据'\t'符进行切片
            line_array = line.strip().split('\t')
            # 添加数据(100个元素排成一行)
            data_mat.append([float(line_array[0]), float(line_array[1])])
            # 添加标签(100个元素排成一行)
            label_mat.append(float(line_array[2]))
        return data_mat, label_mat


    """
    函数说明：通过核函数将数据转换更高维空间
    Parameters:
        X - 数据矩阵
        A - 单个数据的向量
        kTup - 包含核函数信息的元组
    Returns:
        K - 计算的核K
    """
    def kernel_trans(xi, xj, kTup):
        """
        :param kTup: 两维，第一列是字符串，为lin,rbf，如果是rbf，第二列多一个
                                                    sigmma
        :return:
        """
        # 读取X的行列数
        m, n = np.shape(xi)
```

```
    # K初始化为m行1列的零向量
    K = np.mat(np.zeros((m, 1)))
    # 线性核函数只进行内积
    if kTup[0] == 'lin':
        K = xi * xj.T
    # 高斯核函数，根据高斯核函数公式计算
    elif kTup[0] == 'rbf':
        for j in range(m):
            delta_row = xi[j, :] - xj
            K[j] = delta_row * delta_row.T
        K = np.exp(K / (-1 * kTup[1] ** 2))
    else:
        raise NameError('核函数无法识别')
    return K

"""
类说明：维护所有需要操作的值
Parameters:
    data_mat_in - 数据矩阵
    class_cabels - 数据标签
    C - 松弛变量
    toler - 容错率
Returns:
    None
"""
"""
定义一个新的数据结构
"""
class opt_struct:
    def __init__(self, data_mat_in, class_labels, C, toler,kTup):
        # 数据矩阵
        self.X = data_mat_in #传进来的数据
        # 数据标签
        self.label_mat = class_labels
        # 松弛变量
        self.C = C
        # 容错率
        self.tol = toler
        # 矩阵的行数
```

```
        self.m = np.shape(data_mat_in)[0]
        # 根据矩阵行数初始化alphas矩阵，一个m行1列的全零列向量
        self.alphas = np.mat(np.zeros((self.m, 1)))
        # 初始化b参数为0
        self.b = 0
        # 根据矩阵行数初始化误差缓存矩阵，第一列为是否有效标志位，其中0无
                                                效，1有效;第二列为实际的误差
                                                Ei的值
        """
        我们之前的定义为:Ei=gxi-yi
        yi是标签的实际值。
        gx=alpha_i*y_i*x_i.x,就相当于是w.x+b
        因为误差值经常用到，所以希望每次计算后放到一个缓存当中，将ecache一
                                                分为二，第一列是标志位，取值
                                                为0或者1，为1时表示已经算出来
        """
        self.ecache = np.mat(np.zeros((self.m, 2)))
        self.K=np.mat(np.zeros((self.m,self.m)))
        for i in range(self.m):
            self.K[:,i]=kernel_trans(self.X,self.X[i,:],kTup)

"""
函数说明：计算误差
Parameters:
    os - 数据结构
    k - 标号为k的数据
Returns:
    Ek - 标号为k的数据误差
"""
def cal_Ek(os, k):
    # multiply(a,b)就是个乘法，如果a,b是两个数组，那么对应元素相乘
    # .T为转置
    fXk = float(np.multiply(os.alphas, os.label_mat).T * os.K[:, k]+os.b)
    # 计算误差项
    Ek = fXk - float(os.label_mat[k])
    # 返回误差项
    return Ek

"""
```

```
函数说明: 随机选择alpha_j
Parameters:
    i - alpha_i的索引值
    m - alpha参数个数
Returns:
    j - alpha_j的索引值
"""
def select_j_random(i, m):
    j = i
    while(j == i):
        # uniform()方法将随机生成一个实数, 它在[x, y)范围内
        j = int(random.uniform(0, m))
    return j

"""
函数说明: 内循环启发方式2
Parameters:
    i - 标号为i的数据的索引值
    os - 数据结构
    Ei - 标号为i的数据误差
Returns:
    j - 标号为j的数据的索引值
    max_K - 标号为maxK的数据的索引值
    Ej - 标号为j的数据误差
"""
def select_j(i, os, Ei):
    # 初始化
    max_K = -1
    max_delta_E = 0
    Ej = 0
    # 根据Ei更新误差缓存
    os.ecache[i] = [1, Ei]
    # 对一个矩阵.A转换为Array类型
    # 返回误差不为0的数据的索引值
    valid_ecache_list = np.nonzero(os.ecache[:, 0].A)[0]
    # 有不为0的误差
    if(len(valid_ecache_list) > 1):
        # 遍历, 找到最大的Ek
        for k in valid_ecache_list:
```

```
            # 不计算k==i节省时间
            if k == i:
                continue
            # 计算Ek
            Ek = cal_Ek(os, k)
            # 计算|Ei - Ek|
            delta_E = abs(Ei - Ek)
            # 找到maxDeltaE
            if(delta_E > max_delta_E):
                max_K = k
                max_delta_E = delta_E
                Ej = Ek
        # 返回maxK, Ej
        return max_K, Ej
    # 没有不为0的误差
    else:
        # 随机选择alpha_j的索引值
        j = select_j_random(i, os.m)
        # 计算Ej
        Ej = cal_Ek(os, j)
    # 返回j, Ej
    return j, Ej

"""
函数说明: 计算Ek,并更新误差缓存
Parameters:
    oS - 数据结构
    k - 标号为k的数据的索引值
Returns:
    None
"""
def update_Ek(os, k):
    # 计算Ek
    Ek = cal_Ek(os, k)
    # 更新误差缓存
    os.ecache[k] = [1, Ek]

"""
函数说明: 修剪alpha_j
```

```
    Parameters:
        aj - alpha_j值
        H - alpha上限
        L - alpha下限
    Returns:
        aj - alpha_j值
    """
    def clip_alpha(aj, H, L):
        if aj > H:
            aj = H
        if L > aj:
            aj = L
        return aj

    """
    函数说明：优化的SMO算法
    Parameters:
        i - 标号为i的数据的索引值
        oS - 数据结构
    Returns:
        1 - 有任意一对alpha值发生变化
        0 - 没有任意一对alpha值发生变化或变化太小
    """
    def innerL(i, os):
        # 步骤1：计算误差Ei
        Ei = cal_Ek(os, i)
        # 优化alpha,设定一定的容错率
        if((os.label_mat[i] * Ei < -os.tol) and (os.alphas[i] < os.C)) or ((os.
                                                label_mat[i] * Ei > os.tol) and (
                                                os.alphas[i] > 0)):
            # 使用内循环启发方式2选择alpha_j,并计算Ej
            j, Ej = select_j(i, os, Ei)
            # 保存更新前的alpha值，使用深层拷贝
            alpha_i_old = os.alphas[i].copy()
            alpha_j_old = os.alphas[j].copy()
            # 步骤2：计算上界H和下界L
            if(os.label_mat[i] != os.label_mat[j]):
                L = max(0, os.alphas[j] - os.alphas[i])
                H = min(os.C, os.C + os.alphas[j] - os.alphas[i])
```

```
        else:
            L = max(0, os.alphas[j] + os.alphas[i] - os.C)
            H = min(os.C, os.alphas[j] + os.alphas[i])
        if L == H:
            print("L == H")
            return 0
        # 步骤3: 计算eta
        eta = 2.0 * os.K[i, j] - os.K[i, i] - os.K[j, j] #这里的计算就要采
                                                          用核函数了
        if eta >= 0:
            print("eta >= 0")
            return 0
        # 步骤4: 更新alpha_j
        os.alphas[j] -= os.label_mat[j] * (Ei - Ej) / eta
        # 步骤5: 修剪alpha_j
        os.alphas[j] = clip_alpha(os.alphas[j], H, L)
        # 更新Ej至误差缓存
        update_Ek(os, j)
        if(abs(os.alphas[j] - alpha_j_old) < 0.00001):
            print("alpha_j变化太小")
            return 0
        # 步骤6: 更新alpha_i
        os.alphas[i] += os.label_mat[i] * os.label_mat[j] * (alpha_j_old -
                                                          os.alphas[j])
        # 更新Ei至误差缓存
        update_Ek(os, i)
        # 步骤7: 更新b_1和b_2:
        b1 = os.b - Ei - os.label_mat[i] * (os.alphas[i] - alpha_i_old) *
                                                          os.K[i, i] - os.label_mat[j]
                                                          * (os.alphas[j] - alpha_j_old
                                                          ) * os.K[j, i]
        b2 = os.b - Ej - os.label_mat[i] * (os.alphas[i] - alpha_i_old) *
                                                          os.K[i, j] - os.label_mat[j]
                                                          * (os.alphas[j] - alpha_j_old
                                                          ) * os.K[j, j]
        # 步骤8: 根据b_1和b_2更新b
        if(0 < os.alphas[i] < os.C):
            os.b = b1
        elif(0 < os.alphas[j] < os.C):
```

```
            os.b = b2
        else:
            os.b = (b1 + b2) / 2.0
        return 1
    else:
        return 0

"""
函数说明: 完整的线性SMO算法
Parameters:
    data_mat_in - 数据矩阵
    class_labels - 数据标签
    C - 松弛变量
    toler - 容错率
    max_iter - 最大迭代次数
    kTup - 包含核函数信息的元组
Returns:
    os.b - SMO算法计算的b
    os.alphas - SMO算法计算的alphas
"""
def smo_p(data_mat_in, class_labels, C, toler, max_iter, kTup = ('lin', 0))
                                                :
    # 初始化数据结构
    os = opt_struct(np.mat(data_mat_in), np.mat(class_labels).transpose(),
                                                C, toler, kTup)
    # 初始化当前迭代次数
    iter = 0
    entrie_set = True
    alpha_pairs_changed = 0
    # 遍历整个数据集alpha都没有更新或者超过最大迭代次数, 则退出循环
    while(iter < max_iter) and ((alpha_pairs_changed > 0) or (entrie_set)):
        alpha_pairs_changed = 0
        # 遍历整个数据集
        if entrie_set:
            for i in range(os.m):
                # 使用优化的SMO算法
                alpha_pairs_changed += innerL(i, os)
                print("全样本遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (
                                                        iter, i,
```

```
                                                    alpha_pairs_changed))
            iter += 1
        # 遍历非边界值
        else:
            # 遍历不在边界0和C的alpha
            non_nound_i_s = np.nonzero((os.alphas.A > 0) * (os.alphas.A < C
                                                    ))[0]
            for i in non_nound_i_s:
                alpha_pairs_changed += innerL(i, os)
                print("非边界遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (
                                                    iter, i,
                                                    alpha_pairs_changed))
            iter += 1
        # 遍历一次后改为非边界遍历
        if entrie_set:
            entrie_set = False
        # 如果alpha没有更新, 计算全样本遍历
        elif(alpha_pairs_changed == 0):
            entrie_set = True
        print("迭代次数:%d" % iter)
    # 返回SMO算法计算的b和alphas
    return os.b, os.alphas

"""
函数说明: 测试函数
Parameters:
    k1 - 使用高斯核函数的时候表示到达率
Returns:
    None
"""
def test_rbf(k1 = 1.3):
    # 加载训练集
    data_array, label_array = load_dataset('testSetRBF.txt')
    # 根据训练集计算b, alphas
    b, alphas = smo_p(data_array, label_array, 200, 0.0001, 100, ('rbf', k1
                                            ))
    data_mat = np.mat(data_array)
    label_mat = np.mat(label_array).transpose()
    # 获得支持向量
```

```
    svInd = np.nonzero(alphas.A > 0)[0] # 从行维度来描述索引值
    sVs = data_mat[svInd]
    labelSV = label_mat[svInd]
    print("支持向量个数:%d" % np.shape(sVs)[0])
    m, n = np.shape(data_mat)
    error_count = 0
    for i in range(m):
        # 计算各个点的核
        kernel_eval = kernel_trans(sVs, data_mat[i, :], ('rbf', k1))
        # 根据支持向量的点计算超平面，返回预测结果
        predict = kernel_eval.T * np.multiply(labelSV, alphas[svInd]) + b
        # 返回数组中各元素的正负号，用1和-1表示，并统计错误个数
        if np.sign(predict) != np.sign(label_array[i]):
            error_count += 1
    # 打印错误率
    print('训练集错误率:%.2f%%' % ((float(error_count) / m) * 100))
    # 加载测试集
    data_array, label_array = load_dataset('testSetRBF2.txt')
    error_count = 0
    data_mat = np.mat(data_array)
    label_mat = np.mat(label_array).transpose()
    m, n = np.shape(data_mat)
    for i in range(m):
        # 计算各个点的核
        kernel_eval = kernel_trans(sVs, data_mat[i, :], ('rbf', k1))
        # 根据支持向量的点计算超平面，返回预测结果
        predict = kernel_eval.T * np.multiply(labelSV, alphas[svInd]) + b
        # 返回数组中各元素的正负号，用1和-1表示，并统计错误个数
        if np.sign(predict) != np.sign(label_array[i]):
            error_count += 1
    # 打印错误率
    print('测试集错误率:%.2f%%' % ((float(error_count) / m) * 100))

def cal_w_s(alphas, data_mat, class_labels):
    X = np.mat(data_mat)
    label_mat = np.mat(class_labels).transpose()
    m, n = np.shape(X)
    w = np.zeros((n, 1))
    for i in range(m):
```

```
        w += np.multiply(alphas[i] * label_mat[i], X[i, :].T)
    return w

"""
函数说明：数据可视化
Parameters:
    dataMat - 数据矩阵
    labelMat - 数据标签
Returns:
    None
"""
def show_dataset(data_mat, label_mat):
    for path in ['testSetRBF.txt','testSetRBF2.txt']:
        data_mat, label_mat = load_dataset(path)
    # 画图
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cm_dark = mpl.colors.ListedColormap(['b', 'g'])
    ax.scatter(np.array(data_mat)[:, 0], np.array(data_mat)[:, 1], c=np.
                                        array(label_mat).squeeze(), cmap=
                                        cm_dark, s=30)
    b, alphas = smo_p(data_mat, label_mat, 200, 0.0001, 10000, ('rbf', 2))
    w = cal_w_s(alphas, data_mat, label_mat)
    # 画支持向量
    alphas_non_zeros_index = np.where(alphas > 0)
    for i in alphas_non_zeros_index[0]:
        circle = Circle((data_mat[i][0], data_mat[i][1]), 0.035, facecolor=
                                            'none', edgecolor='red',
                                            linewidth=1.5,alpha=1)
        ax.add_patch(circle)
    plt.show()

if __name__ == '__main__':
    for path in ['testSetRBF.txt','testSetRBF2.txt']:
        data_mat, label_mat = load_dataset(path)
        print('=='*60)
        show_dataset(data_mat,label_mat)
        print('=='*60)
    test_rbf()
```

4. 示例：手写识别问题回顾

```
# -*- coding: utf-8 -*-
import numpy as np
import random
import operator
"""
函数说明：读取数据
Parameters:
    file_name - 文件名
Returns:
    data_mat - 数据矩阵
    label_mat - 数据标签
"""
def load_dataset(file_name):
    # 数据矩阵
    data_mat = []
    # 标签向量
    label_mat = []
    # 打开文件
    fr = open(file_name)
    # 逐行读取
    for line in fr.readlines():
        # 去掉每一行首尾的空白符，例如'\n','\r','\t',' '
        # 将每一行内容根据'\t'符进行切片
        line_array = line.strip().split('\t')
        # 添加数据(100个元素排成一行)
        data_mat.append([float(line_array[0]), float(line_array[1])])
        # 添加标签(100个元素排成一行)
        label_mat.append(float(line_array[2]))
    return data_mat, label_mat


"""
函数说明：通过核函数将数据转换更高维空间
Parameters:
    X - 数据矩阵
    A - 单个数据的向量
    kTup - 包含核函数信息的元组
Returns:
```

```
    K - 计算的核K
"""
def kernel_trans(xi, xj, kTup):
    """
    :param kTup: 两维，第一列是字符串，为lin,rbf，如果是rbf，第二列多一个
                                                    sigmma
    :return:
    """
    # 读取X的行列数
    m, n = np.shape(xi)
    # K初始化为m行1列的零向量
    K = np.mat(np.zeros((m, 1)))
    # 线性核函数只进行内积
    if kTup[0] == 'lin':
        K = xi * xj.T
    # 高斯核函数，根据高斯核函数公式计算
    elif kTup[0] == 'rbf':
        for j in range(m):
            delta_row = xi[j, :] - xj
            K[j] = delta_row * delta_row.T
        K = np.exp(K / (-1 * kTup[1] ** 2))
    else:
        raise NameError('核函数无法识别')
    return K


"""
类说明：维护所有需要操作的值
Parameters:
    dataMatIn - 数据矩阵
    classLabels - 数据标签
    C - 松弛变量
    toler - 容错率
Returns:
    None
"""
"""
定义一个新的数据结构
"""
class opt_struct:
```

```
    def __init__(self, data_mat_in, class_labels, C, toler,kTup):
        # 数据矩阵
        self.X = data_mat_in #传进来的数据
        # 数据标签
        self.label_mat = class_labels
        # 松弛变量
        self.C = C
        # 容错率
        self.tol = toler
        # 矩阵的行数
        self.m = np.shape(data_mat_in)[0]
        # 根据矩阵行数初始化alphas矩阵，一个m行1列的全零列向量
        self.alphas = np.mat(np.zeros((self.m, 1)))
        # 初始化b参数为0
        self.b = 0
        # 根据矩阵行数初始化误差缓存矩阵，第一列为是否有效标志位，其中0无
                                                效，1有效;第二列为实际的误差
                                                Ei的值
        """
        我们之前的定义为:Ei=gxi-yi
        yi是标签的实际值。
        gx=alpha_i*y_i*x_i.x,就相当于是w.x+b
        因为误差值经常用到，所以希望每次计算后放到一个缓存当中，将ecache一
                                                分为二，第一列是标志位，取值
                                                为0或者1，为1时表示已经算出来
        """
        self.ecache = np.mat(np.zeros((self.m, 2)))
        self.K=np.mat(np.zeros((self.m,self.m)))
        for i in range(self.m):
            self.K[:,i]=kernel_trans(self.X,self.X[i,:],kTup)

"""
函数说明：计算误差
Parameters:
    os - 数据结构
    k - 标号为k的数据
Returns:
    Ek - 标号为k的数据误差
"""
```

```
def cal_Ek(os, k):
    # multiply(a,b)就是个乘法，如果a,b是两个数组，那么对应元素相乘
    # .T为转置
    fXk = float(np.multiply(os.alphas, os.label_mat).T * os.K[:, k]+os.b)
    # 计算误差项
    Ek = fXk - float(os.label_mat[k])
    # 返回误差项
    return Ek

"""
函数说明：随机选择alpha_j
Parameters:
    i - alpha_i的索引值
    m - alpha参数个数
Returns:
    j - alpha_j的索引值
"""
def select_j_random(i, m):
    j = i
    while(j == i):
        # uniform()方法将随机生成一个实数，它在[x, y)范围内
        j = int(random.uniform(0, m))
    return j

"""
函数说明：内循环启发方式2
Parameters:
    i - 标号为i的数据的索引值
    os - 数据结构
    Ei - 标号为i的数据误差
Returns:
    j - 标号为j的数据的索引值
    max_K - 标号为maxK的数据的索引值
    Ej - 标号为j的数据误差
"""
def select_j(i, os, Ei):
    # 初始化
    max_K = -1
    max_delta_E = 0
```

```
    Ej = 0
    # 根据Ei更新误差缓存
    os.ecache[i] = [1, Ei]
    # 对一个矩阵.A转换为Array类型
    # 返回误差不为0的数据的索引值
    valid_ecache_list = np.nonzero(os.ecache[:, 0].A)[0]
    # 有不为0的误差
    if(len(valid_ecache_list) > 1):
        # 遍历, 找到最大的Ek
        for k in valid_ecache_list:
            # 不计算k==i节省时间
            if k == i:
                continue
            # 计算Ek
            Ek = cal_Ek(os, k)
            # 计算|Ei - Ek|
            delta_E = abs(Ei - Ek)
            # 找到maxDeltaE
            if(delta_E > max_delta_E):
                max_K = k
                max_delta_E = delta_E
                Ej = Ek
        # 返回maxK, Ej
        return max_K, Ej
    # 没有不为0的误差
    else:
        # 随机选择alpha_j的索引值
        j = select_j_random(i, os.m)
        # 计算Ej
        Ej = cal_Ek(os, j)
    # 返回j, Ej
    return j, Ej

"""
函数说明: 计算Ek,并更新误差缓存
Parameters:
    oS - 数据结构
    k - 标号为k的数据的索引值
Returns:
```

```
    None
"""
def update_Ek(os, k):
    # 计算Ek
    Ek = cal_Ek(os, k)
    # 更新误差缓存
    os.ecache[k] = [1, Ek]

"""
函数说明: 修剪alpha_j
Parameters:
    aj - alpha_j值
    H - alpha上限
    L - alpha下限
Returns:
    aj - alpha_j值
"""
def clip_alpha(aj, H, L):
    if aj > H:
        aj = H
    if L > aj:
        aj = L
    return aj

"""
函数说明: 优化的SMO算法
Parameters:
    i - 标号为i的数据的索引值
    oS - 数据结构
Returns:
    1 - 有任意一对alpha值发生变化
    0 - 没有任意一对alpha值发生变化或变化太小
"""
def innerL(i, os):
    # 步骤1: 计算误差Ei
    Ei = cal_Ek(os, i)
    # 优化alpha,设定一定的容错率
    if((os.label_mat[i] * Ei < -os.tol) and (os.alphas[i] < os.C)) or ((os.
                                            label_mat[i] * Ei > os.tol) and (
```

```
                                        os.alphas[i] > 0)):
    # 使用内循环启发方式2选择alpha_j,并计算Ej
    j, Ej = select_j(i, os, Ei)
    # 保存更新前的alpha值, 使用深层拷贝
    alpha_i_old = os.alphas[i].copy()
    alpha_j_old = os.alphas[j].copy()
    # 步骤2: 计算上界H和下界L
    if(os.label_mat[i] != os.label_mat[j]):
        L = max(0, os.alphas[j] - os.alphas[i])
        H = min(os.C, os.C + os.alphas[j] - os.alphas[i])
    else:
        L = max(0, os.alphas[j] + os.alphas[i] - os.C)
        H = min(os.C, os.alphas[j] + os.alphas[i])
    if L == H:
        #print("L == H")
        return 0
    # 步骤3: 计算eta
    eta = 2.0 * os.K[i, j] - os.K[i, i] - os.K[j, j] #这里的计算就要采
                                                      用核函数了
    if eta >= 0:
        #print("eta >= 0")
        return 0
    # 步骤4: 更新alpha_j
    os.alphas[j] -= os.label_mat[j] * (Ei - Ej) / eta
    # 步骤5: 修剪alpha_j
    os.alphas[j] = clip_alpha(os.alphas[j], H, L)
    # 更新Ej至误差缓存
    update_Ek(os, j)
    if(abs(os.alphas[j] - alpha_j_old) < 0.00001):
        #print("alpha_j变化太小")
        return 0
    # 步骤6: 更新alpha_i
    os.alphas[i] += os.label_mat[i] * os.label_mat[j] * (alpha_j_old -
                                                      os.alphas[j])
    # 更新Ei至误差缓存
    update_Ek(os, i)
    # 步骤7: 更新b_1和b_2:
    b1 = os.b - Ei - os.label_mat[i] * (os.alphas[i] - alpha_i_old) *
                                                      os.K[i, i] - os.label_mat[j]
```

```
                                                * (os.alphas[j] - alpha_j_old
                                                ) * os.K[j, i]
            b2 = os.b - Ej - os.label_mat[i] * (os.alphas[i] - alpha_i_old) *
                                                os.K[i, j] - os.label_mat[j]
                                                * (os.alphas[j] - alpha_j_old
                                                ) * os.K[j, j]
            # 步骤8: 根据b_1和b_2更新b
            if(0 < os.alphas[i] < os.C):
                os.b = b1
            elif(0 < os.alphas[j] < os.C):
                os.b = b2
            else:
                os.b = (b1 + b2) / 2.0
            return 1
        else:
            return 0

"""
函数说明: 完整的线性SMO算法
Parameters:
    data_mat_in - 数据矩阵
    class_labels - 数据标签
    C - 松弛变量
    toler - 容错率
    max_iter - 最大迭代次数
    kTup - 包含核函数信息的元组
Returns:
    os.b - SMO算法计算的b
    os.alphas - SMO算法计算的alphas
"""
def smo_p(data_mat_in, class_labels, C, toler, max_iter, kTup = ('lin', 0))
                                                :
    # 初始化数据结构
    os = opt_struct(np.mat(data_mat_in), np.mat(class_labels).transpose(),
                                                C, toler, kTup)
    # 初始化当前迭代次数
    iter = 0
    entrie_set = True
    alpha_pairs_changed = 0
```

```
    # 遍历整个数据集alpha都没有更新或者超过最大迭代次数，则退出循环
    while(iter < max_iter) and ((alpha_pairs_changed > 0) or (entrie_set)):
        alpha_pairs_changed = 0
        # 遍历整个数据集
        if entrie_set:
            for i in range(os.m):
                # 使用优化的SMO算法
                alpha_pairs_changed += innerL(i, os)
                #print("全样本遍历:第%d次迭代 样本:%d, alpha优化次数:%d" %
                                                    (iter, i,
                                                    alpha_pairs_changed))
            iter += 1
        # 遍历非边界值
        else:
            # 遍历不在边界0和C的alpha
            non_nound_i_s = np.nonzero((os.alphas.A > 0) * (os.alphas.A < C
                                                    ))[0]
            for i in non_nound_i_s:
                alpha_pairs_changed += innerL(i, os)
                #print("非边界遍历:第%d次迭代 样本:%d, alpha优化次数:%d" %
                                                    (iter, i,
                                                    alpha_pairs_changed))
            iter += 1
        # 遍历一次后改为非边界遍历
        if entrie_set:
            entrie_set = False
        # 如果alpha没有更新，计算全样本遍历
        elif(alpha_pairs_changed == 0):
            entrie_set = True
        print("迭代次数:%d" % iter)
    # 返回SMO算法计算的b和alphas
    return os.b, os.alphas


"""
函数说明：将32*32的二进制图像转换为1*1024向量
Parameters:
    filename - 文件名
Returns:
```

```
    returnVect - 返回二进制图像的1*1024向量
"""
def imgage_vector(file_name):
    # 创建1*1024零向量
    return_vector = np.zeros((1, 1024))
    # 打开文件
    fr = open(file_name)
    # 按行读取
    for i in range(32):
        # 读取一行数据
        line_string = fr.readline()
        # 每一行的前32个数据依次存储到returnVect中
        for j in range(32):
            return_vector[0, 32*i+j] = int(line_string[j])
    # 返回转换后的1*1024向量
    return return_vector


"""
函数说明：加载图片
Parameters:
    dirName - 文件夹名字
Returns:
    trainingMat - 数据矩阵
    data_labels - 数据标签
"""
def load_images(dir_name):
    from os import listdir
    # 测试集的Labels
    data_labels = []
    # 返回trainingDigits目录下的文件名
    training_file_list = listdir(dir_name)
    # 返回文件夹下文件的个数
    m = len(training_file_list)
    # 初始化训练的Mat矩阵（全零阵），测试集
    training_mat = np.zeros((m, 1024))
    # 从文件名中解析出训练集的类别
    for i in range(m):
        # 获得文件的名字
```

```
        file_name_string = training_file_list[i]
        file_string = file_name_string.split('.')[0]
        # 获得分类的数字
        class_number = int(file_string.split('_')[0])
        if class_number == 9:
            data_labels.append(-1)
        else:
            data_labels.append(1)
        training_mat[i, :] = imgage_vector('%s/%s' % (dir_name,
                                                      file_name_string))
    return training_mat, data_labels

"""
函数说明: 测试函数
Parameters:
    kTup - 包含核函数信息的元组
Returns:
    None
"""
def test_digits(kTup=('rbf', 10)):
    # 加载训练集
    data_array, label_array = load_images('trainingDigits')
    # 根据训练集计算b, alphas
    b, alphas = smo_p(data_array, label_array, 200, 0.001, 10, kTup)
    data_mat = np.mat(data_array)
    label_mat = np.mat(label_array).transpose()
    # 获得支持向量
    svInd = np.nonzero(alphas.A > 0)[0]
    sVs = data_mat[svInd]
    labelSV = label_mat[svInd]
    print("支持向量个数:%d" % np.shape(sVs)[0])
    m, n = np.shape(data_mat)
    error_count = 0
    for i in range(m):
        # 计算各个点的核
        kernel_eval = kernel_trans(sVs, data_mat[i, :], kTup)
        # 根据支持向量的点计算超平面, 返回预测结果
        predict = kernel_eval.T * np.multiply(labelSV, alphas[svInd]) + b
        # 返回数组中各元素的正负号, 用1和-1表示, 并统计错误个数
```

```
        if np.sign(predict) != np.sign(label_mat[i]):
            error_count += 1
    # 打印错误率
    print('训练集错误率:%.2f%%' % (float(error_count) / m))
    # 加载测试集
    data_array, label_array = load_images('testDigits')
    error_count = 0
    data_mat = np.mat(data_array)
    label_mat = np.mat(label_array).transpose()
    m, n = np.shape(data_mat)
    for i in range(m):
        # 计算各个点的核
        kernel_eval = kernel_trans(sVs, data_mat[i, :], kTup)
        # 根据支持向量的点计算超平面，返回预测结果
        predict = kernel_eval.T * np.multiply(labelSV, alphas[svInd]) + b
        # 返回数组中各元素的正负号，用1和-1表示，并统计错误个数
        if np.sign(predict) != np.sign(label_array[i]):
            error_count += 1
    # 打印错误率
    print('测试集错误率:%.2f%%' % (float(error_count) / m))

"""
KNN算法
"""
def classify(test_dataset,train_dataset,labels,k):
    """
    :param test_dataset:用于分类的数据（测试集）
    :param train_dataset:用于训练的数据（训练集）
    :param labels:分类标签
    :param k:选择距离最小的k个点
    :return:
    """
    # numpy函数shape[0]返回dataset的行数
    dataset_size=train_dataset.shape[0]
    # tile具有重复功能，dataset_size是重复四遍，后面的1保证重复完了是四行，
                                            而不是一行里有四个是一样的
    diff_mat=np.tile(test_dataset,(dataset_size,1))-train_dataset
    # 二次特征相减后平方
    sqr_diff_mat=diff_mat**2
```

```
    # sum()所有元素相加，sum(0）列相加，sum(1)行相加
    sq_distance=sqr_diff_mat.sum(axis=1)
    # 开方，计算出距离
    distance=sq_distance**0.5
    # 返回distance中元素从小到大排列后的索引值
    sorted_distance=distance.argsort()
    # 定一个记录类别次数的字典
    class_count={}
    for i in range(k):
        # 取出前k个元素的类别
        label=labels[sorted_distance[i]]
        # dict.get(key,default=None)，字典的get()方法，返回指定键的值，如果
                                                值不在字典中返回默认值
        # 计算类别次数
        class_count[label]=class_count.get(label,0)+1
        # python3中用items()替换python2中的iteritems()
        # key=operator.iteritems(1)根据字典的值进行排序
        # key=operator.iteritems(0)根据字典的键进行排序
        # reverse降序排列字典
    sorted_class_count=sorted(class_count.items(),key=operator.itemgetter(1
                                            ),reverse=True)
    # 返回次数最多的类别
    return sorted_class_count[0][0]

def hand_writing_class_test():
    import os
    data_labels = []
    # 导入训练集，‘trainingDigits’是一个文件夹
    training_file_list = os.listdir('trainingDigits')
    # 计算训练样本个数
    m = len(training_file_list)
    # 初始化数据集，将所有训练数据用一个m行，1024列的矩阵表示
    training_mat = np.zeros((m,1024))
    for i in range(m):
        # 获得所有文件名，文件名格式‘x_y.txt’,x表示这个手写数字实际表示的
                                                数字（label）
        file_name_string = training_file_list[i]
        # 去除 .txt
        file_str = file_name_string.split('.')[0]
```

```
        # classnumber为每个样本的分类，用‘_’分割，取得label
        class_num_str = int(file_str.split('_')[0])
        # 将所有标签都存进hwLables[]
        data_labels.append(class_num_str)
        # 将文件转化为向量后存入trainingMat[],这里展现了灵活的文件操作
        training_mat[i,:] = imgage_vector('./trainingDigits/%s' %
                                                file_name_string)
    test_file_list = os.listdir('testDigits')  #迭代测试集
    error_count = 0.0
    m_test = len(test_file_list)
    for i in range(m_test):
        file_name_string = test_file_list[i]
        # 去除 .txt
        file_str = file_name_string.split('.')[0]
        class_num_string = int(file_str.split('_')[0])
        # 这部分针对测试集的预处理和前面基本相同
        vector_under_test = imgage_vector('./testDigits/%s' %
                                                file_name_string)
        # 使用算法预测样本所属类别，调用了前面写的classify0()函数
        classify_result = classify(vector_under_test, training_mat,
                                                data_labels, 3)
        # print ("分类结果:%d,真实类别:%d" % (classify_result,
                                                class_num_string))
        # 算法结果与样本的实际分类做对比
        if (classify_result != class_num_string): error_count += 1.0
    print('KNN算法的预测情况:')
    print ("\n分类错误的个数为: %d" % error_count)
    print ("\n分类错误率为: %f" % (error_count/float(m_test)))

if __name__ == '__main__':
    test_digits()
    hand_writing_class_test()
```