



# 机器学习

## 课程实践报告

项目名称: 支持向量机 SMO 算法理论、实现与应用

学 号: 220120175011

学生姓名: 张忆

所在学院: 工商管理学院

专 业: 大数据管理

# 目录

|                               |          |
|-------------------------------|----------|
| <b>1 前言 .....</b>             | <b>1</b> |
| 1.1 支持向量机概述 .....             | 1        |
| 1.2 支持向量机算法概述 .....           | 2        |
| 1.2.1 支持向量机算法分类 .....         | 3        |
| 1.2.2 SMO 算法的提出 .....         | 3        |
| <b>2 支持向量机 SMO 算法理论 .....</b> | <b>4</b> |
| 2.1 SMO 算法的基本思路 .....         | 4        |
| 2.2 两个变量二次规划的求解方法 .....       | 5        |
| 2.2.1 子问题的表达式 .....           | 5        |
| 2.2.2 子问题的约束条件 .....          | 5        |
| 2.2.3 计算最优解 .....             | 6        |
| 2.3 变量的选择方法 .....             | 7        |
| 2.3.1 第 1 个变量的选择（外层循环） .....  | 7        |
| 2.3.2 第 2 个变量的选择（内层循环） .....  | 7        |
| 2.3.3 计算阈值和差值 .....           | 7        |
| 2.4 SMO 算法 .....              | 8        |
| <b>3 支持向量机 SMO 算法实现 .....</b> | <b>9</b> |
| 3.1 简化版算法 .....               | 9        |
| 3.1.1 算法思路 .....              | 9        |
| 3.1.2 算法实现 .....              | 9        |
| 3.1.3 实验结果 .....              | 10       |
| 3.2 完整版算法 .....               | 12       |
| 3.2.1 改进之处 .....              | 12       |
| 3.2.2 算法实现 .....              | 12       |
| 3.2.3 实验结果 .....              | 13       |

|  |           |
|--|-----------|
| 3.3 核函数 .....                          | 14        |
| 3.3.1 RBF 核函数.....                     | 14        |
| 3.3.2 算法实现 .....                       | 15        |
| 3.3.3 实验结果 .....                       | 15        |
| <b>4 支持向量机 SMO 算法应用 .....</b>          | <b>17</b> |
| 4.1 求解 MNIST 问题 .....                  | 17        |
| 4.1.1 查看数据 .....                       | 17        |
| 4.1.2 数据预处理 .....                      | 18        |
| 4.1.3 构建模型 .....                       | 19        |
| 4.1.4 训练模型 .....                       | 19        |
| 4.1.5 评估模型 .....                       | 19        |
| 4.2 与 MNIST 官方数据比较 .....               | 20        |
| 4.3 与 sci-kit learn 比较 .....           | 21        |
| <b>参考文献.....</b>                       | <b>22</b> |
| <b>附录.....</b>                         | <b>23</b> |
| 附件一：SMO 简化版代码 .....                    | 23        |
| 附件二：SMO 完整版代码 .....                    | 31        |
| 附件三：加入核函数的 SMO 完整版代码 .....             | 42        |
| 附件四：使用附件三的 SMO 完整版代码求解 MNIST 问题.....   | 55        |
| 附件五：使用 sci-kit learn 求解 MNIST 问题 ..... | 69        |

# 1 前言

## 1.1 支持向量机概述

支持向量机 (support vector machines, SVM) 是一种二类分类模型。它的基本模型是定义在特征空间上的间隔最大的线性分类器，间隔最大使它有别于感知机；支持向量机还包括核技巧，这使它成为实质上的非线性分类器。支持向量机的学习策略就是间隔最大化，可形式化为一个求解凸二次规划 (convex quadratic programming) 的问题，也等价于正则化的合页损失函数的最小化问题。支持向量机的学习算法是求解凸二次规划的最优化算法。

支持向量机学习方法包含构建由简至繁的模型：线性可分支持向量机 (linear support vector machine in linearly separable case)、线性支持向量机 (linear support vector machine) 以及非线性支持向量机 (non-linear support vector machine)。简单模型是复杂模型的基础，也是复杂模型的特殊情况。当训练数据线性可分时，通过硬间隔最大化 (hard margin maximization)，学习一个线性的分类器，即线性可分支持向量机，又称为硬间隔支持向量机；当训练数据近似线性可分时，通过软间隔最大化 (soft margin maximization)，也学习一个线性的分类器，即线性支持向量机，又称为软间隔支持向量机；当训练数据线性不可分时，通过使用核技巧 (kernel trick) 及软间隔最大化，学习非线性支持向量机。

当输入空间为欧氏空间或离散集合、特征空间为希尔伯特空间时，核函数 (kernel function) 表示将输入从输入空间映射到特征空间得到的特征向量之间的内积。通过使用核函数可以学习非线性支持向量机，等价于隐式地在高维的特征空间中学习线性支持向量机。这样的方法称为核技巧。核方法 (kernel method) 是比支持向量机更为一般的机器学习方法。

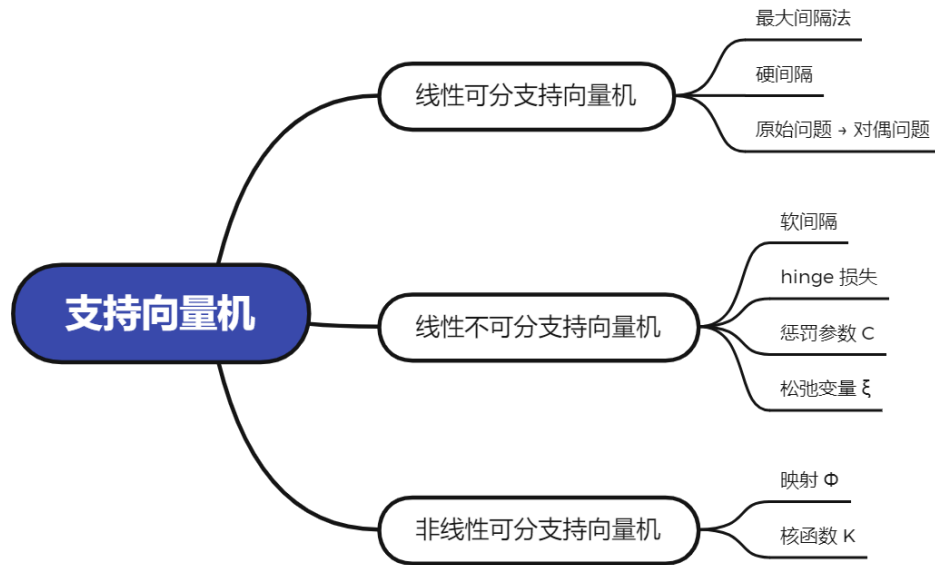


图 1 支持向量机知识结构网络图

## 1.2 支持向量机算法概述

支持向量机主要是求解如下的凸二次规划的对偶问题：

$$\begin{aligned}
 \min_{\alpha} \quad & \frac{1}{2} \sum_{i=1}^l \sum_{j=1}^l y_i y_j \alpha_i \alpha_j K(x_i, x_j) - \sum_{i=1}^l \alpha_i, \\
 \text{s.t.} \quad & \sum_{i=1}^l y_i \alpha_i = 0, \\
 & 0 \leq \alpha_i \leq C, \quad i = 1, \dots, l.
 \end{aligned}$$

这个问题可表述为如下的紧凑形式：

$$\begin{aligned}
 \min_{\alpha} \quad & d(\alpha) = \frac{1}{2} \alpha^T H \alpha - e^T \alpha, \\
 \text{s.t.} \quad & y^T \alpha = 0, \\
 & 0 \leq \alpha \leq C e,
 \end{aligned}$$

虽然一般的求解二次规划的算法原则上可以直接求解该问题，但实际上在处理大型问题时，由于存储和计算量两方面的要求，这些算法往往会失

效，因此求解支持向量机二次规划问题产生了单独的算法。

### 1.2.1 支持向量机算法分类

支持向量机算法主要分为两类：选块算法和分解算法。

#### (1) 选块算法

选块算法的“块”指的是训练集  $T$  中称为工作集的子集。所谓“选块”就是通过某种迭代方式逐步排除块中非支持向量对应的训练点，并逐步把所有支持向量对应的训练点选入块中。选块算法是一种启发式算法，缺点是当支持向量很多时，选块算法会遇到所需存储量过大的困难。

#### (2) 分解算法

分解算法的特点是每次更新若干个（一定数量的） $\alpha$  的分量，而其他的分量则保持不变。准备更新的  $\alpha$  的分量对应的训练点组成的集合，就是当前的工作集。迭代过程中只是将当前工作集之外的训练点中的一部分“情况最糟的点”与工作集中的同等数量的训练点进行交换，工作集的规模是固定不变的。

### 1.2.2 SMO 算法的提出

SMO（Sequential Minimal Optimization，序列最小最优化）算法在 1998 年由微软研究院研究员 John Platt 提出。与传统求解凸二次规划问题的算法相比，SMO 算法计算速度快，能大幅减少计算时间。

SMO 算法是分解算法的一种特殊情形。与通常的分解算法比较，尽管它可能需要更多的迭代次数，但是由于每次迭代的计算量很少，该算法常表现出整体的快速收敛性质。另外，该算法还具有其他一些重要优点，如不需要储存核矩阵、没有矩阵运算、容易实现等。

## 2 支持向量机 SMO 算法理论

### 2.1 SMO 算法的基本思路

SMO 算法要解如下凸二次规划的对偶问题：

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j K(x_i, x_j) - \sum_{i=1}^N \alpha_i \\ \text{s.t.} \quad & \sum_{i=1}^N \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C, i = 1, 2, \dots, N \end{aligned}$$

在这个问题中，变量是拉格朗日乘子，一个变量  $\alpha_i$  对应于一个样本点  $(x_i, y_i)$ ；变量的总数等于训练样本容量  $N$ 。

SMO 算法是一种启发式算法，其基本思路是：如果所有变量的解都满足此最优化问题的 KKT 条件（Karush-Kuhn-Tucker conditions），那么这个最优化问题的解就得到了。否则，选择两个变量，固定其他变量，针对这两个变量构建一个二次规划子问题，这个子问题关于这两个变量的解应该更接近原始二次规划问题的解。子问题有两个变量，其中一个变量是违反 KKT 条件最严重的，另一个由约束条件自动确定。子问题可以通过解析方法求解。

若选择  $\alpha_1, \alpha_2$  为目标变量，固定  $\alpha_3, \alpha_4, \dots, \alpha_N$ ，由等式约束  $\sum_{i=1}^N \alpha_i y_i = 0$  可知

$$\alpha_1 = -y_1 \sum_{i=2}^N \alpha_i y_i, \text{ 因此选择的两个变量中只有一个是自由变量。}$$

按照上述基本思路，SMO 算法将原问题不断分解为子问题并对子问题求解，进而达到求解原问题的目的。

整个 SMO 算法包括两个部分：求解两个变量二次规划的解析方法和选择变量的启发式方法。

## 2.2 两个变量二次规划的求解方法

### 2.2.1 子问题的表达式

选择  $\alpha_1, \alpha_2$  为目标变量，固定  $\alpha_3, \alpha_4, \dots, \alpha_N$ ，则最优化问题子问题的表达式为：

$$\begin{aligned} \min_{\alpha_1, \alpha_2} W(\alpha_1, \alpha_2) &= \frac{1}{2} K_{11} \alpha_1^2 + \frac{1}{2} K_{22} \alpha_2^2 + y_1 y_2 K_{12} \alpha_1 \alpha_2 - \\ &\quad (\alpha_1 + \alpha_2) + y_1 \alpha_1 \sum_{i=3}^N y_i \alpha_i K_{i1} + y_2 \alpha_2 \sum_{i=3}^N y_i \alpha_i K_{i2} \\ \text{s.t.} \quad &\alpha_1 y_1 + \alpha_2 y_2 = - \sum_{i=3}^N y_i \alpha_i = C_1 \\ &0 \leq \alpha_i \leq C, i = 1, 2 \end{aligned}$$

其中  $K_{ij} = K(i, j), i, j = 1, 2, \dots, N$ ，该目标函数中省略了不含  $\alpha_1, \alpha_2$  的常数项。

### 2.2.2 子问题的约束条件

$\alpha_1, \alpha_2$  两个变量的约束可以用二维空间中的图形表示：

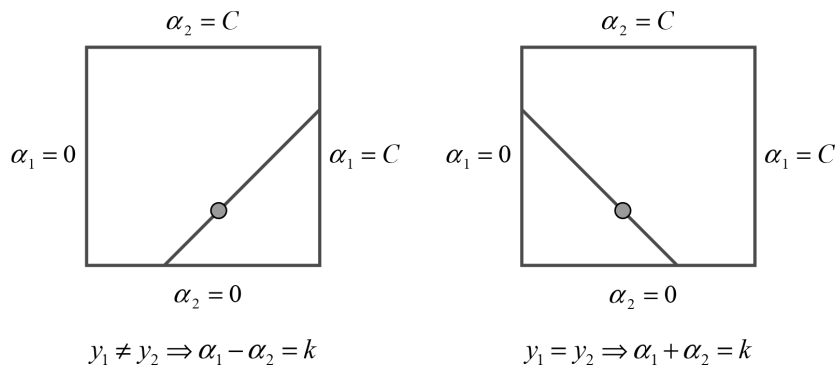


图2 二变量优化问题图示

不等式约束  $0 \leq \alpha_i \leq C$  使得  $\alpha_1, \alpha_2$  在盒子  $[0, C] \times [0, C]$  内，等式约束



$\alpha_1 y_1 + \alpha_2 y_2 = -\sum_{i=3}^N y_i \alpha_i = C_1$  使得  $\alpha_1, \alpha_2$  在平行于盒子  $[0, C] \times [0, C]$  的对角线的直线上，因此要求的是目标函数在一条平行于对角线的线段上的最优值。这使得两个变量的最优化问题成为实质上的单变量的最优化问题，不妨考虑为变量  $\alpha_2$  的最优化问题。

### 2.2.3 计算最优解

设最优化问题的初始可行解为  $\alpha_1^{old}, \alpha_2^{old}$ ，最优解为  $\alpha_1^{new}, \alpha_2^{new}$ ，并且假设在沿着约束方向未经剪辑时  $\alpha_2$  的最优解为  $\alpha_2^{new,unc}$ 。由于  $\alpha_2^{new}$  需满足不等式约束  $0 \leq \alpha_i \leq C$ ，所以  $\alpha_2^{new}$  的取值范围为  $L \leq \alpha_2^{new} \leq H$ ，其中  $L$  与  $H$  是  $\alpha_2^{new}$  所在的对角线段端点的界。

如果  $y_1 \neq y_2$ ，则  $L = \max(0, \alpha_2^{old} - \alpha_1^{old})$ ,  $H = \min(C, C + \alpha_2^{old} - \alpha_1^{old})$ ；如果  $y_1 = y_2$ ，则  $L = \max(0, \alpha_2^{old} + \alpha_1^{old} - C)$ ,  $H = \min(C, \alpha_2^{old} + \alpha_1^{old})$ 。

记  $g(x) = \sum_{i=1}^N \alpha_i y_i K(x_i, x) + b$ ， $E_i = g(x_i) - y_i = \left( \sum_{j=1}^N \alpha_j y_j K(x_j, x_i) + b \right) - y_i$ ，当  $i=1, 2$  时， $E_i$  为  $g(x)$  对  $x_i$  的预测值与真实输出  $y_i$  之差。

沿着约束方向未经剪辑时的解  $\alpha_2^{new,unc} = \alpha_2^{old} + \frac{y_2(E_1 - E_2)}{\eta}$ ，其中  $\eta = K_{11} + K_{22} - 2K_{12} = \|\Phi(x_1) - \Phi(x_2)\|^2$ ， $\Phi(x)$  是输入空间到特征空间的映射。

经剪辑后的解为：

$$\alpha_2^{new} = \begin{cases} H, & \alpha_2^{new,unc} > H \\ \alpha_2^{new,unc}, & L \leq \alpha_2^{new,unc} \leq H \\ L, & \alpha_2^{new,unc} < L \end{cases}$$

并求出  $\alpha_1^{new} = \alpha_1^{old} + y_1 y_2 (\alpha_2^{old} - \alpha_2^{new})$ 。

## 2.3 变量的选择方法

SMO 算法在每个子问题中选择两个变量优化，其中至少一个变量是违反 KKT 条件的。

### 2.3.1 第 1 个变量的选择（外层循环）

外层循环在训练样本中选取违反 KKT 条件最严重的样本点，并将其对应的变量作为第 1 个变量，即检验训练样本点  $(x_i, y_i)$  是否满足 KKT 条件，即：

$$\begin{aligned}\alpha_i = 0 &\Leftrightarrow y_i g(x_i) \geq 1 \\ 0 < \alpha_i < C &\Leftrightarrow y_i g(x_i) = 1 \\ \alpha_i = C &\Leftrightarrow y_i g(x_i) \leq 1\end{aligned}$$

$$\text{其中 } g(x_i) = \sum_{j=1}^N \alpha_j y_j K(x_i, x_j) + b。$$

在检验过程中，外层循环首先遍历所有满足条件  $0 \leq \alpha_i \leq C$  的样本点，即在间隔边界上的支持向量点，检验它们是否满足 KKT 条件。如果这些样本点都满足 KKT 条件，那么遍历整个训练集，检验它们是否满足 KKT 条件。

### 2.3.2 第 2 个变量的选择（内层循环）

假设在外层循环中已经找到第 1 个变量  $\alpha_1$ ，现在要在内层循环中找第 2 个变量  $\alpha_2$ 。第 2 个变量选择的标准是希望能使  $\alpha_2$  有足够大的变化。一种简单的做法是选择  $\alpha_2$ ，使其对应的  $|E_1 - E_2|$  最大。因为  $\alpha_1$  已定， $E_1$  也确定。如果  $E_1$  为正，选择最小的  $E_i$  作为  $E_2$ ；如果  $E_1$  为负，选择最大的  $E_i$  作为  $E_2$ 。

### 2.3.3 计算阈值和差值

在每次完成两个变量的优化后，都要重新计算阈值  $b$ 。

经计算可得阈值为：

$$b_1^{new} = -E_1 - y_1 K_{11}(\alpha_1^{new} - \alpha_1^{old}) - y_2 K_{21}(\alpha_2^{new} - \alpha_2^{old}) + b^{old}$$

$$b_2^{new} = -E_2 - y_1 K_{12}(\alpha_1^{new} - \alpha_1^{old}) - y_2 K_{22}(\alpha_2^{new} - \alpha_2^{old}) + b^{old}$$

差值为：

$$E_i^{new} = \sum_S y_j \alpha_j K(x_i, x_j) + b^{new} - y_i$$

其中  $S$  是所有支持向量  $x_j$  的集合。

## 2.4 SMO 算法

SMO 算法流程如下：

输入：训练数据集  $T = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$ ，其中  $x_i \in X = R^n$ ， $y_i \in Y = \{-1, +1\}$ ， $i = 1, 2, \dots, N$ ，精度  $\varepsilon$ ；

输出：近似解  $\hat{\alpha}$ 。

(1) 取初值  $\alpha^{(0)} = 0$ ，令  $k = 0$ ；

(2) 选取优化变量  $\alpha_1^{(k)}, \alpha_2^{(k)}$ ，解析求解两个变量的最优化问题，求得最优解  $\alpha_1^{(k+1)}, \alpha_2^{(k+1)}$ ，更新  $\alpha$  为  $\alpha^{(k+1)}$ ；

(3) 若在精度  $\varepsilon$  范围内满足停机条件

$$\sum_{i=1}^N \alpha_i y_i = 0, \quad 0 \leq \alpha_i \leq C, \quad i = 1, 2, \dots, N$$

$$y_i \cdot g(x_i) = \begin{cases} \geq 1, & \{x_i \mid \alpha_i = 0\} \\ = 1, & \{x_i \mid 0 < \alpha_i < C\} \\ \leq 1, & \{x_i \mid \alpha_i = C\} \end{cases}$$

$$g(x_i) = \sum_{j=1}^N \alpha_j y_j K(x_j, x_i) + b$$

则转 (4)；否则令  $k = k + 1$ ，转 (2)；

(4) 取  $\hat{\alpha} = \alpha^{(k+1)}$ 。

## 3 支持向量机 SMO 算法实现

本章首先根据 SMO 算法的基本工作思路给出简化版算法，再通过优化得到完整版算法，最后加入核函数，得到能用于处理非线性可分数据的完整版算法。

### 3.1 简化版算法

#### 3.1.1 算法思路

简化版算法如下所示：

```
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
    如果所有向量都没被优化，增加迭代数目，继续下一次循环
```

图 3 简化版 SMO 算法伪代码

#### 3.1.2 算法实现

（1）loadDataSet 函数

功能：读取并处理数据

输入：数据集

处理：对数据集的数据去除空格、切片

输出：数据矩阵、数据标签

（2）selectJrand 函数

功能：随机选择  $\alpha_j$

输入：  $\alpha_i$

处理：随机选择  $\alpha_j$ ，使得  $\alpha_j \neq \alpha_i$

输出：  $\alpha_j$

### （3）clipAlpha 函数

功能：判断并调整  $\alpha_j$  的取值

输入：  $\alpha_j$ 、 $\alpha_j$  的上限  $H$ 、下限  $L$

处理：若  $\alpha_j < L$  则  $\alpha_j = L$ ，若  $\alpha_j > H$  则  $\alpha_j = H$

输出：  $\alpha_j$

### （4）smoSimple 函数

功能：简化版 SMO 算法计算  $\alpha$ ， $b$

输入：数据矩阵，数据标签，惩罚参数，容错率，最大迭代次数

处理：按照 3.1.1 节简化版 SMO 算法进行计算

输出：  $\alpha$ ， $b$

### （5）get\_w 函数

功能：计算直线法向量  $w$

输入：数据矩阵，数据标签， $\alpha$

处理：根据公式计算

输出：  $w$

### （6）showClassifier 函数

功能：分类结果可视化

输入：数据矩阵、 $w$ 、 $b$

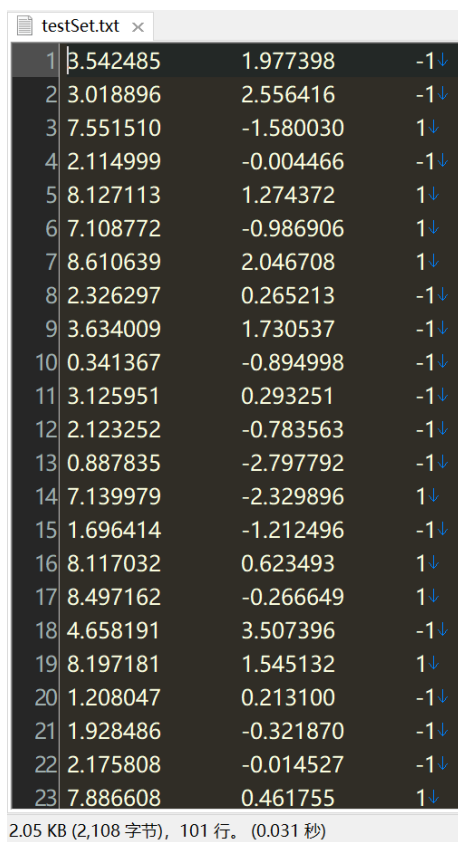
处理：绘制样本散点图、直线、标记支持向量

输出：可视化结果

简化版算法完整代码见附件一。

## 3.1.3 实验结果

数据集：100 条 2 维数据，每条数据均有标签。



|    |          |           |     |
|----|----------|-----------|-----|
| 1  | 3.542485 | 1.977398  | -1↓ |
| 2  | 3.018896 | 2.556416  | -1↓ |
| 3  | 7.551510 | -1.580030 | 1↓  |
| 4  | 2.114999 | -0.004466 | -1↓ |
| 5  | 8.127113 | 1.274372  | 1↓  |
| 6  | 7.108772 | -0.986906 | 1↓  |
| 7  | 8.610639 | 2.046708  | 1↓  |
| 8  | 2.326297 | 0.265213  | -1↓ |
| 9  | 3.634009 | 1.730537  | -1↓ |
| 10 | 0.341367 | -0.894998 | -1↓ |
| 11 | 3.125951 | 0.293251  | -1↓ |
| 12 | 2.123252 | -0.783563 | -1↓ |
| 13 | 0.887835 | -2.797792 | -1↓ |
| 14 | 7.139979 | -2.329896 | 1↓  |
| 15 | 1.696414 | -1.212496 | -1↓ |
| 16 | 8.117032 | 0.623493  | 1↓  |
| 17 | 8.497162 | -0.266649 | 1↓  |
| 18 | 4.658191 | 3.507396  | -1↓ |
| 19 | 8.197181 | 1.545132  | 1↓  |
| 20 | 1.208047 | 0.213100  | -1↓ |
| 21 | 1.928486 | -0.321870 | -1↓ |
| 22 | 2.175808 | -0.014527 | -1↓ |
| 23 | 7.886608 | 0.461755  | 1↓  |

2.05 KB (2,108 字节), 101 行。 (0.031 秒)

图 4 简化版 SMO 算法实验数据集

超参数设置：

- 惩罚参数=0.6
- 容错率=0.001
- 最大迭代次数=40

运行时间=3.28 s

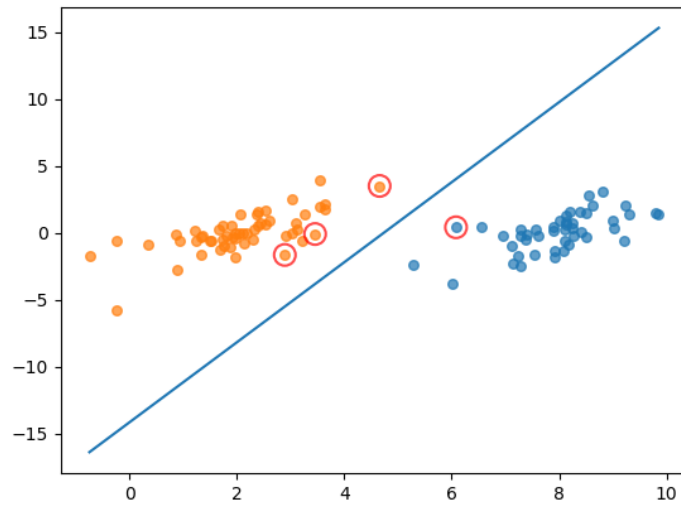


图 5 简化版 SMO 算法实验结果

## 3.2 完整版算法

在小规模数据集上，简化版算法可以正常运行，但在更大的数据集上运行速度会变慢。完整版算法应用了一些能够提速的启发式方法。

### 3.2.1 改进之处

完整版算法在简化版算法的基础上做了如下优化：

- (1) 建立一个类用于计算误差、存储误差缓存；
- (2) 选择使误差改变最大的  $\alpha$  作为  $\alpha_j$ ；
- (3) 修改循环判断条件，缩短运行时间。

经过优化后的完整版算法，不仅能用于更大规模的数据集，运行速度也更快。

### 3.2.2 算法实现

- (1) optStruct 类

功能：存储所有需要操作的值

输入：数据矩阵、数据标签、惩罚参数、容错率、 $\alpha$ 、 $b$ 、误差缓存

#### (2) calcEk 函数

功能：计算误差

输入：第  $k$  个数据

处理：根据公式计算

输出：数据误差

#### (3) selectJ 函数

功能：启发式方法选择  $\alpha_j$

输入： $\alpha_i$ ， $E_i$

处理：选择使得  $|E_i - E_k|$  最大的  $\alpha_j$

输出： $\alpha_j$

#### (4) updateEk 函数

功能：更新误差缓存

#### (5) innerL 函数

功能：判断有无一对  $\alpha$  发生变化，有变化则输出 1，没有变化则输出 0

#### (6) smoP 函数

功能：完整版 SMO 算法计算  $\alpha$ ， $b$

输入：数据矩阵，数据标签，惩罚参数，容错率，最大迭代次数

处理：按照完整版 SMO 算法进行计算

输出： $\alpha$ ， $b$

完整版算法完整代码见附件二。

### 3.2.3 实验结果

数据集：与 3.1.3 节数据集相同。

超参数设置：与 3.1.3 节超参数设置相同。

运行时间=0.20 s

与简化版算法相比，在相同的数据集上设置相同的超参数，完整版算法



运行速度明显提高。

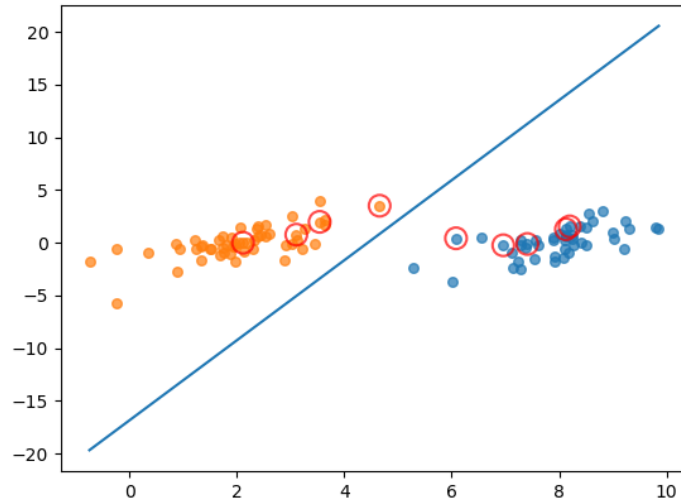


图 6 完整版 SMO 算法实验结果

### 3.3 核函数

针对数据非线性可分的情况，引入核函数将数据映射到高维空间，使得数据线性可分。使用核函数不需知道映射  $\Phi(x)$  的具体形式和所属空间，并且把计算高维向量的内积转化为低维向量的计算问题，避免了维数灾难的问题。

#### 3.3.1 RBF 核函数

RBF (Radial Basis Function, 径向基函数) 核函数是支持向量机中最为常用的核函数，定义如下：

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

其中  $\sigma$  是可以自定义的超参数。

### 3.3.2 算法实现

#### (1) kernelTrans 函数

功能：通过核函数将数据转换到高维空间

输入：数据矩阵，数据向量，核函数类型

处理：根据公式计算线性核函数、RBF 核函数

输出：计算结果

加入核函数的完整版算法完整代码见附件三。

### 3.3.3 实验结果

数据集：训练集：100 条 2 维数据；测试集：100 条 2 维数据。训练集和测试集均有标签。

| testSetRBF.txt |           |           |            |
|----------------|-----------|-----------|------------|
| 1              | -0.214824 | 0.662756  | -1.000000↓ |
| 2              | -0.061569 | -0.091875 | 1.000000↓  |
| 3              | 0.406933  | 0.648055  | -1.000000↓ |
| 4              | 0.223650  | 0.130142  | 1.000000↓  |
| 5              | 0.231317  | 0.766906  | -1.000000↓ |
| 6              | -0.748800 | -0.531637 | -1.000000↓ |
| 7              | -0.557789 | 0.375797  | -1.000000↓ |
| 8              | 0.207123  | -0.019463 | 1.000000↓  |
| 9              | 0.286462  | 0.719470  | -1.000000↓ |
| 10             | 0.195300  | -0.179039 | 1.000000↓  |

| testSetRBF2.txt |           |           |            |
|-----------------|-----------|-----------|------------|
| 1               | 0.676771  | -0.486687 | -1.000000↓ |
| 2               | 0.008473  | 0.186070  | 1.000000↓  |
| 3               | -0.727789 | 0.594062  | -1.000000↓ |
| 4               | 0.112367  | 0.287852  | 1.000000↓  |
| 5               | 0.383633  | -0.038068 | 1.000000↓  |
| 6               | -0.927138 | -0.032633 | -1.000000↓ |
| 7               | -0.842803 | -0.423115 | -1.000000↓ |
| 8               | -0.003677 | -0.367338 | 1.000000↓  |
| 9               | 0.443211  | -0.698469 | -1.000000↓ |
| 10              | -0.473835 | 0.005233  | 1.000000↓  |

图 7 加入核函数的完整版 SMO 算法实验数据集

#### (1) 实验一

超参数设置：

- 惩罚参数=200
- 容错率=0.0001
- 最大迭代次数=100
- 核函数类型=RBF
- 核函数参数=0.1

实验结果：

- 训练集错误率=0.00%

- 测试集错误率=8.00%
- 支持向量个数=88
- 运行时间=1.23 s

## (2) 实验二

超参数设置:

- 惩罚参数=200
- 容错率=0.0001
- 最大迭代次数=100
- 核函数类型=RBF
- 核函数参数=1.3

实验结果:

- 训练集错误率=0.00%
- 测试集错误率=2.00%
- 支持向量个数=26
- 运行时间=0.32 s

对比实验一和实验二, 在固定惩罚参数、容错率、最大迭代次数、核函数类型四个超参数, 只改变核函数参数的情况下, 考察核函数参数 $\sigma$ 对分类结果的影响。当 $\sigma$ 由 0.1 变为 1.3 后, 运行时间减少, 支持向量个数减少, 但测试集错误率上升。

## 4 支持向量机 SMO 算法应用

本章使用 3.3 节加入核函数的完整版算法求解 MNIST 问题，将结果与 MNIST 官方数据比较，再使用 `sci-kit learn` 求解相同问题并比较结果。

### 4.1 求解 MNIST 问题

MNIST 问题是一个手写数字识别问题，数据集来自美国国家标准与技术研究所，由来自 250 个不同的人手写的数字构成，每一张图片包含对应的标签。MNIST 问题是机器学习的经典入门问题，可使用多种机器学习模型加以训练，并预测图片中的数字。

#### 4.1.1 查看数据

原始的 MNIST 数据集中，训练集有 60,000 条数据，测试集有 10,000 条数据。每张图片大小为  $28 \times 28$ ，像素取值为  $[0, 1]$ ，能表示灰度。考虑到 3.3 节加入核函数的完整版算法性能有限，而且只能处理二分类问题，因此对原始数据进行简化，只取出数字 1 和数字 7 两种标签的图片，399 条数据作为训练集，193 条数据作为测试集。图片大小更改为  $32 \times 32$ ，像素取值为 0 或 1，不表示灰度。

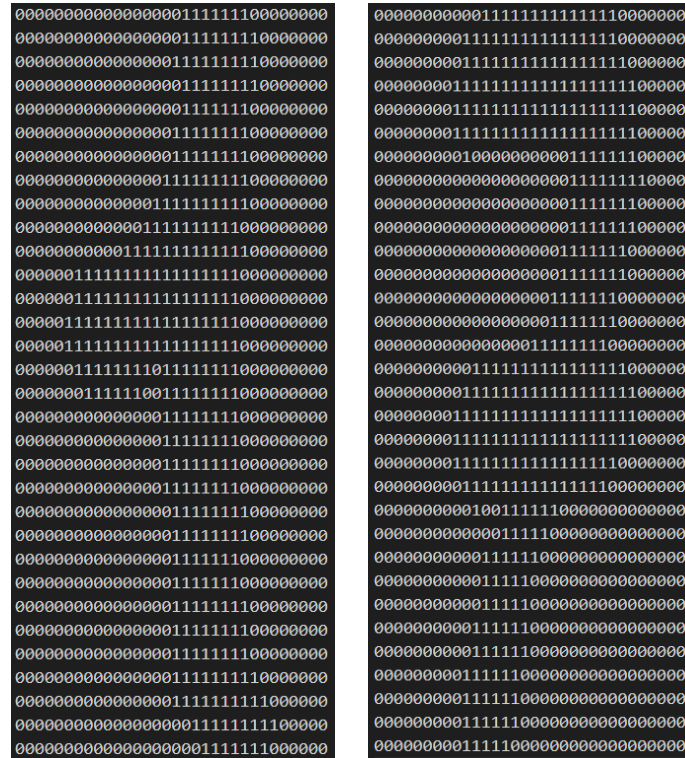


图 8 简化后的 MNIST 实验数据集

#### 4.1.2 数据预处理

将每张图片的 32 行 $\times$ 32 列数组转换为 1 行 $\times$ 1024 列数组。

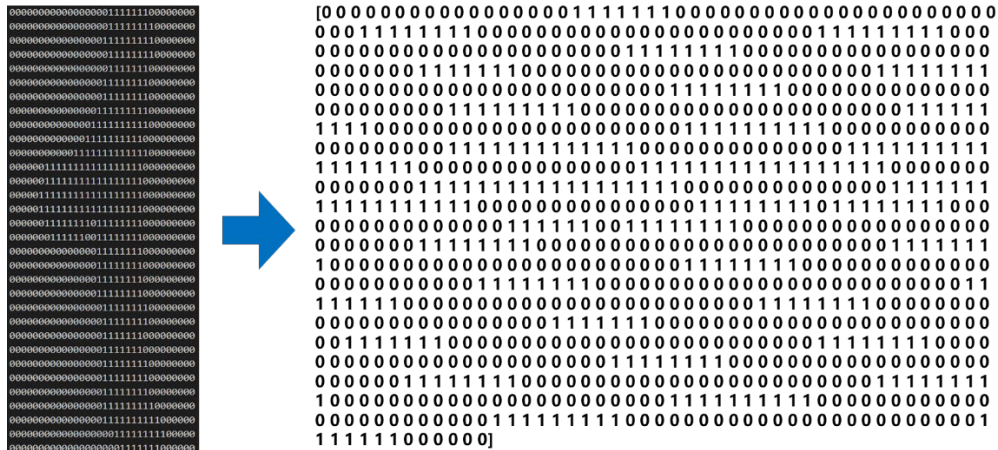


图 9 数据预处理

### 4.1.3 构建模型

使用 3.3 节加入核函数的完整版算法构建模型，完整代码见附件四。

### 4.1.4 训练模型

训练模型有以下 5 个步骤：

- (1) 加载训练集
- (2) 设置超参数
- (3) 计算  $\alpha$  ,  $b$
- (4) 计算支持向量
- (5) 生成决策函数

### 4.1.5 评估模型

超参数设置：

- 惩罚参数=200
- 容错率=0.0001
- 最大迭代次数=10000

选择核函数类型为 RBF 和 Linear，对 RBF 核函数，设置核函数参数为 0.1, 5, 10, 50, 100，分别进行实验，得到以下实验结果：

**表 1 采用不同的核函数和超参数的 MNIST 实验结果**

| 核函数    | 超参数 | 训练错误率 | 测试错误率  | 支持向量数 | 训练用时  | 测试用时 |
|--------|-----|-------|--------|-------|-------|------|
| RBF    | 0.1 | 0.00% | 50.26% | 399   | 11.67 | 1.17 |
| RBF    | 5   | 0.00% | 0.52%  | 399   | 24.59 | 1.18 |
| RBF    | 10  | 0.00% | 0.52%  | 93    | 6.64  | 0.33 |
| RBF    | 50  | 0.50% | 1.04%  | 29    | 3.71  | 0.15 |
| RBF    | 100 | 1.25% | 2.07%  | 25    | 5.17  | 0.16 |
| Linear | 0   | 2.01% | 3.63%  | 21    | 2.36  | 0.10 |

从实验结果可看出，选择 RBF 核函数，超参数从 0.1 逐渐增大到 100 时，训练错误率逐渐增大，测试错误率先减小后增大，支持向量数逐渐减小，测试用时逐渐减小，训练用时变化不规律，先减小后增大再减小。

当超参数为 0.1 和 5 时，支持向量数等于训练样本数，相当于把所有训练样本均视为支持向量。当超参数为 10 时，训练错误率和测试错误率均为最小。

当选择 Linear 核函数时，训练错误率和测试错误率没有明显高于选择 RBF 核函数时的最优值，并且测试用时比选择 RBF 核函数时所有情况都要小，因此可以认为，当数据量很大时，选择 Linear 核函数，以牺牲错误率来换取分类速度的提高，也是一种可以接受的方案。

## 4.2 与 MNIST 官方数据比较

MNIST 官方网站发布了使用各种机器学习、深度学习模型进行分类的结果，使用支持向量机的分类结果如下表所示<sup>①</sup>：

**表 2 MNIST 官方发布采用 SVM 的分类结果**

| CLASSIFIER                                | PREPROCESSING | TEST ERROR RATE (%) | Reference                         |
|---|---------------|---------------------|-----------------------------------|
| <b>SVMs</b>                               |               |                     |                                   |
| SVM, Gaussian Kernel                      | none          | 1.4                 |                                   |
| SVM deg 4 polynomial                      | deskewing     | 1.1                 | <a href="#">LeCun et al. 1998</a> |
| Reduced Set SVM deg 5 polynomial          | deskewing     | 1.0                 | <a href="#">LeCun et al. 1998</a> |
| Virtual SVM deg-9 poly [distortions]      | none          | 0.8                 | <a href="#">LeCun et al. 1998</a> |
| Virtual SVM, deg-9 poly, 1-pixel jittered | none          | 0.68                | DeCoste and Scholkopf, MLJ 2002   |
| Virtual SVM, deg-9 poly, 1-pixel jittered | deskewing     | 0.68                | DeCoste and Scholkopf, MLJ 2002   |
| Virtual SVM, deg-9 poly, 2-pixel jittered | deskewing     | 0.56                | DeCoste and Scholkopf, MLJ 2002   |

可以看出，本文所构建的支持向量机分类模型与官方数据相比，测试错误率较为接近。但 4.1 节的实验是将原始数据进行了简化处理的，数据量也大幅缩减，而官方发布的数据是用完整的数据集分类得出的结果，因此无法准确判断本文所构建的支持向量机分类模型的性能与其他支持向量机分类模型有多大的差距。

<sup>①</sup> THE MNIST DATABASE of handwritten digits, <http://yann.lecun.com/exdb/mnist/>

### 4.3 与 sci-kit learn 比较

为了准确测试本文所构建的支持向量机分类模型的性能，使用 sci-kit learn 机器学习库中的 svm.SVC 分类模型再次进行实验。

数据集：与 4.1 节数据集相同。

超参数设置：与 4.1 节超参数设置相同。

- 惩罚参数=200
- 容错率=0.0001
- 最大迭代次数=10000

完整代码见附件五。

实验结果：

- 测试集错误率=0.52%
- 训练用时=0.17 s
- 测试用时=0.10 s

从实验结果可看出，本文所构建的支持向量机分类模型的性能与 sci-kit learn 机器学习库的支持向量机分类模型还有一定差距。通过查阅 sci-kit learn 官方文档可知，sci-kit learn 机器学习库的支持向量机分类模型使用的是 LIBSVM。LIBSVM 使用二阶信息选取工作集，并且使用 C++语言编写，因此分类速度更快。<sup>②</sup>

---

<sup>②</sup> LIBSVM -- A Library for Support Vector Machines, <https://www.csie.ntu.edu.tw/~cjlin/libsvm/>



## 参考文献

- [1] 李航著. 统计学习方法 第 2 版[M]. 北京: 清华大学出版社, 2019.05.
- [2] 邓乃扬, 田英杰著. 支持向量机: 理论、算法与拓展[M]. 北京: 科学出版社, 2009.08.
- [3] (美) PeterHarrington; 李锐, 李鹏, 曲亚东, 王斌译者. 机器学习实战[M]. 北京: 人民邮电出版社, 2013.06.

## 附录

### 附件一：SMO 简化版代码

```
# -*- coding: utf-8 -*-
"""
简化版 SMO 算法
"""

import matplotlib.pyplot as plt
import numpy as np
import random
import time

def loadDataSet(fileName):
    """
    读取数据

    Parameters:
        fileName - 文件名

    Returns:
        dataMat - 数据矩阵
        labelMat - 数据标签
    """

    # 数据矩阵
    dataMat = []
    # 标签向量
    labelMat = []

    # 打开文件
    fr = open(fileName)

    # 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
        lineArr = line.strip().split('\t')
```

```
# 添加数据(100 个元素排成一行)
dataMat.append([float(lineArr[0]), float(lineArr[1])])

# 添加标签(100 个元素排成一行)
labelMat.append(float(lineArr[2]))

return dataMat, labelMat


def selectJrand(i, m):
    """
    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
    """

    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


def clipAlpha(aj, H, L):
    """
    修剪 alpha_j

    Parameters:
        aj - alpha_j 值
        H - alpha 上限
        L - alpha 下限

    Returns:
        aj - alpha_j 值
    """
```

```
if aj > H:
    aj = H

if L > aj:
    aj = L

return aj


def smoSimple(dataMatIn, classLabels, C, toler, maxIter):
    """
    计算 alpha, b

    Parameters:
        dataMatIn - 数据矩阵
        classLabels - 数据标签
        C - 惩罚参数
        toler - 容错率
        maxIter - 最大迭代次数

    Returns:
        None
    """

    # 转换为 numpy 的 mat 矩阵存储(100,2)
    dataMatrix = np.mat(dataMatIn)
    # 转换为 numpy 的 mat 矩阵存储并转置(100,1)
    labelMat = np.mat(classLabels).transpose()

    # 初始化 b 参数, 设为 0
    b = 0
    # 统计 dataMatrix 的维度,m:100 行; n:2 列
    m, n = np.shape(dataMatrix)
    # 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 < maxIter:
        # 初始优化次数
        alphaPairsChanged = 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
    fxi = float(np.multiply(alphas, labelMat).T * (dataMatrix *
dataMatrix[i, :].T)) + b
    # 误差项计算公式
    Ei = fxi - float(labelMat[i])

    # 优化 alpha, 设定一定的容错率
    if ((labelMat[i] * Ei < -toler) and (alphas[i] < C)) or ((labelMat[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 = selectJrand(i, m)

        # 步骤 1, 计算误差 Ej
        fxj = float(np.multiply(alphas, labelMat).T * (dataMatrix *
dataMatrix[j, :].T)) + b
        # 误差项计算公式
        Ej = fxj - float(labelMat[j])

        # 保存更新前的 alpha 值
        alphaIold = alphas[i].copy()
        alphaJold = alphas[j].copy()

        # 步骤 2: 计算上下界 H 和 L
        if labelMat[i] != labelMat[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
        eta = 2.0 * dataMatrix[i, :] * dataMatrix[j, :].T - dataMatrix[i, :] *
dataMatrix[i, :].T - dataMatrix[j, :] * dataMatrix[j, :].T

        if eta >= 0:
            print("eta>=0")
```

```

        continue

    # 步骤 4: 更新 alpha_j
    alphas[j] -= labelMat[j] * (Ei - Ej) / eta

    # 步骤 5: 修剪 alpha_j
    alphas[j] = clipAlpha(alphas[j], H, L)

    if abs(alphas[j] - alphaJold) < 0.00001:
        print("alpha_j 变化太小")
        continue

    # 步骤 6: 更新 alpha_i
    alphas[i] += labelMat[j] * labelMat[i] * (alphaJold - alphas[j])

    # 步骤 7: 更新 b_1 和 b_2
    b1 = b - Ei - labelMat[i] * (alphas[i] - alphaIold) * dataMatrix[i, :] *
dataMatrix[i, :].T - labelMat[j] * (alphas[j] - alphaJold) * dataMatrix[j, :] *
dataMatrix[i, :].T
    b2 = b - Ej - labelMat[i] * (alphas[i] - alphaIold) * dataMatrix[i, :] *
dataMatrix[j, :].T - labelMat[j] * (alphas[j] - alphaJold) * dataMatrix[j, :] *
dataMatrix[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
    alphaPairsChanged += 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, alphaPairsChanged))

    # 更新迭代次数
    if (alphaPairsChanged == 0):
        iter_num += 1
    else:
        iter_num = 0
    print("迭代次数: %d" % iter_num)

```

```
    return b, alphas

def get_w(dataMat, labelMat, alphas):
    """
    计算 w

    Returns:
        dataMat - 数据矩阵
        labelMat - 数据标签
        alphas - alphas 值

    Returns:
        w - 直线法向量
    """

    alphas, dataMat, labelMat = np.array(alphas), np.array(dataMat),
    np.array(labelMat)
    # 通过 labelMat.reshape(1, -1), Numpy 自动计算出有 100 行, 新的数组
    shape 属性为(100, 1)
    # np.tile(labelMat.reshape(1, -1).T, (1, 2))将 labelMat 扩展为两列(将第 1 列复
    制得到第 2 列)
    # w = sum(alpha_i * yi * xi)
    w = np.dot((np.tile(labelMat.reshape(1, -1).T, (1, 2)) * dataMat).T, alphas)

    return w.tolist()

def showClassifier(dataMat, w, b):
    """
    分类结果可视化

    Parameters:
        dataMat - 数据矩阵
        w - 直线法向量
        b - 直线截距

    Returns:
        None
    """

    # 正样本
    data_plus = []
```

---

```

# 负样本
data_minus = []

for i in range(len(dataMat)):
    if labelMat[i] > 0:
        data_plus.append(dataMat[i])
    else:
        data_minus.append(dataMat[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
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
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(dataMat)[0]
x2 = min(dataMat)[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
for i, alpha in enumerate(alphas):
    # 支持向量机的点
    if abs(alpha) > 0:
        x, y = dataMat[i]
        plt.scatter([x], [y], s=150, c='none', alpha=0.7, linewidth=1.5,
edgecolors='red')

plt.show()

if __name__ == '__main__':

```



```
# 开始计时
time_start = time.time()

dataMat, labelMat = loadDataSet('testSet.txt')
b, alphas = smoSimple(dataMat, labelMat, 0.6, 0.001, 40)
w = get_w(dataMat, labelMat, alphas)
showClassifier(dataMat, w, b)

# 结束计时
time_end = time.time()
print('运行时间: ', time_end-time_start)
```

## 附件二：SMO 完整版代码

```
# -*- coding: utf-8 -*-
"""
完整版 SMO 算法
"""

import matplotlib.pyplot as plt
import numpy as np
import random
import time

class optStruct:
    """
    存储所有需要操作的值

    Parameters:
        dataMatIn - 数据矩阵
        classLabels - 数据标签
        C - 惩罚参数
        toler - 容错率

    Returns:
        None
    """

    def __init__(self, dataMatIn, classLabels, C, toler):

        # 数据矩阵
        self.X = dataMatIn
        # 数据标签
        self.labelMat = classLabels
        # 惩罚参数
        self.C = C
        # 容错率
        self.tol = toler
        # 矩阵的行数
        self.m = np.shape(dataMatIn)[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，第二列为实际的误差 E 的值

```
self.eCache = np.mat(np.zeros((self.m, 2)))
```

```
def loadDataSet(fileName):
```

```
    """
```

```
    读取数据
```

```
    Parameters:
```

```
        fileName - 文件名
```

```
    Returns:
```

```
        dataMat - 数据矩阵
```

```
        labelMat - 数据标签
```

```
    """
```

```
# 数据矩阵
```

```
dataMat = []
```

```
# 标签向量
```

```
labelMat = []
```

```
# 打开文件
```

```
fr = open(fileName)
```

```
# 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
```

```
    lineArr = line.strip().split('\t')
```

```
    # 添加数据(100 个元素排成一行)
```

```
    dataMat.append([float(lineArr[0]), float(lineArr[1])])
```

```
    # 添加标签(100 个元素排成一行)
```

```
    labelMat.append(float(lineArr[2]))
```

```
return dataMat, labelMat
```

```
def calcEk(oS, k):
```

```
    """
```

```
    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
"""

fXk = float(np.multiply(oS.alphas, oS.labelMat).T * (oS.X * oS.X[k, :].T) +
oS.b)

# 计算误差项
Ek = fXk - float(oS.labelMat[k])

return Ek

def selectJrand(i, m):
    """
    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
    """

    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

def selectJ(i, oS, Ei):
    """
    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

maxK - 标号为 maxK 的数据的索引值

Ej - 标号为 j 的数据误差

"""

# 初始化

maxK = -1

maxDeltaE = 0

Ej = 0

# 根据 Ei 更新误差缓存

oS.eCache[i] = [1, Ei]

# 返回误差不为 0 的数据的索引值

validEcacheList = np.nonzero(oS.eCache[:, 0].A)[0]

# 有不为 0 的误差

if len(validEcacheList) > 1:

    # 遍历，找到最大的 Ek

    for k in validEcacheList:

        # 不计算 k==i 节省时间

        if k == i:

            continue

        # 计算 Ek

        Ek = calcEk(oS, k)

        # 计算|Ei - Ek|

        deltaE = abs(Ei - Ek)

        # 找到 maxDeltaE

        if deltaE > maxDeltaE:

            maxK = k

            maxDeltaE = deltaE

            Ej = Ek

return maxK, Ej

---

```

# 没有不为 0 的误差
else:
    # 随机选择 alpha_j 的索引值
    j = selectJrand(i, oS.m)

    # 计算 Ej
    Ej = calcEk(oS, j)

return j, Ej

def updateEk(oS, k):
    """
    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
    """

    # 计算 Ek
    Ek = calcEk(oS, k)

    # 更新误差缓存
    oS.eCache[k] = [1, Ek]

def clipAlpha(aj, H, L):
    """
    修剪 alpha_j

    Parameters:
        aj - alpha_j 值
        H - alpha 上限
        L - alpha 下限

    Returns:
        aj - alpha_j 值
    """

    if aj > H:

```

```
    aj = H

    if L > aj:
        aj = L

    return aj

def innerL(i, oS):
    """
    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
    """

    # 步骤 1: 计算误差 Ei
    Ei = calcEk(oS, i)

    # 优化 alpha, 设定一定的容错率
    if ((oS.labelMat[i] * Ei < -oS.tol) and (oS.alphas[i] < oS.C)) or ((oS.labelMat[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 = selectJ(i, oS, Ei)

        # 保存更新前的 alpha 值, 使用深层拷贝
        alphaIold = oS.alphas[i].copy()
        alphaJold = oS.alphas[j].copy()

        # 步骤 2: 计算上界 H 和下界 L
        if oS.labelMat[i] != oS.labelMat[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.labelMat[j] * (Ei - Ej) / eta

# 步骤 5: 修剪 alpha_j
oS.alphas[j] = clipAlpha(oS.alphas[j], H, L)

# 更新 Ej 至误差缓存
updateEk(oS, j)

if abs(oS.alphas[j] - alphaJold) < 0.00001:
    print("alpha_j 变化太小")

    return 0

# 步骤 6: 更新 alpha_i
oS.alphas[i] += oS.labelMat[i] * oS.labelMat[j] * (alphaJold - oS.alphas[j])

# 更新 Ei 至误差缓存
updateEk(oS, i)

# 步骤 7: 更新 b_1 和 b_2:
b1 = oS.b - Ei - oS.labelMat[i] * (oS.alphas[i] - alphaIold) * oS.X[i, :] *
oS.X[i, :].T - oS.labelMat[j] * (oS.alphas[j] - alphaJold) * oS.X[j, :] * oS.X[i, :].T
b2 = oS.b - Ej - oS.labelMat[i] * (oS.alphas[i] - alphaIold) * oS.X[i, :] *
oS.X[j, :].T - oS.labelMat[j] * (oS.alphas[j] - alphaJold) * 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
else:
    return 0

def smoP(dataMatIn, classLabels, C, toler, maxIter):
    """
    完整的线性 SMO 算法

    Parameters:
        dataMatIn - 数据矩阵
        classLabels - 数据标签
        C - 惩罚参数
        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
    """

    # 初始化数据结构
    oS = optStruct(np.mat(dataMatIn), np.mat(classLabels).transpose(), C, toler)

    # 初始化当前迭代次数
    iter = 0
    entrieSet = True
    alphaPairsChanged = 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 < maxIter) and ((alphaPairsChanged > 0) or (entrieSet)):
        alphaPairsChanged = 0

        # 遍历整个数据集
        if entrieSet:

            for i in range(oS.m):
                # 使用优化的 SMO 算法
                alphaPairsChanged += 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, alphaPairsChanged))
    iter += 1

    # 遍历非边界值
    else:
        # 遍历不在边界 0 和 C 的 alpha
        nonBoundIs = np.nonzero((oS.alphas.A > 0) * (oS.alphas.A < C))[0]

        for i in nonBoundIs:
            alphaPairsChanged += 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, alphaPairsChanged))
            iter += 1

            # 遍历一次后改为非边界遍历
            if entrieSet:
                entrieSet = False

            # 如果 alpha 没有更新, 计算全样本遍历
            elif(alphaPairsChanged == 0):
                entrieSet = True
            print("迭代次数:%d" % iter)

        return oS.b, oS.alphas

def calcWs(alphas, dataArr, classLabels):
    """
    计算 w

    Parameters:
        dataArr - 数据矩阵
        classLabels - 数据标签
        alphas - alphas 值

    Returns:
        w - 直线法向量
    """

    X = np.mat(dataArr)
    labelMat = np.mat(classLabels).transpose()
    m, n = np.shape(X)
    w = np.zeros((n, 1))

```

```
    for i in range(m):
        w += np.multiply(alphas[i] * labelMat[i], X[i, :].T)

    return w

def showClassifier(dataMat, classLabels, w, b):
    """
    分类结果可视化

    Parameters:
        dataMat - 数据矩阵
        classLabels - 数据标签
        w - 直线法向量
        b - 直线截距

    Returns:
        None
    """

    # 正样本
    data_plus = []
    # 负样本
    data_minus = []

    for i in range(len(dataMat)):
        if classLabels[i] > 0:
            data_plus.append(dataMat[i])
        else:
            data_minus.append(dataMat[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
    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
    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(dataMat)[0]
x2 = min(dataMat)[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
for i, alpha in enumerate(alphas):
    # 支持向量机的点
    if abs(alpha) > 0:
        x, y = dataMat[i]
        plt.scatter([x], [y], s=150, c='none', alpha=0.7, linewidth=1.5,
edgecolors='red')

plt.show()

if __name__ == '__main__':

    # 开始计时
    time_start = time.time()

    dataArr, classLabels = loadDataSet('testSet.txt')
    b, alphas = smoP(dataArr, classLabels, 0.6, 0.001, 40)
    w = calcWs(alphas, dataArr, classLabels)
    showClassifier(dataArr, classLabels, w, b)

    # 结束计时
    time_end = time.time()
    print('运行时间: ', time_end - time_start)
```

## 附件三：加入核函数的 SMO 完整版代码

```
# -*- coding: utf-8 -*-
"""
完整版 SMO 算法 + 核函数
"""

import matplotlib.pyplot as plt
import numpy as np
import random
import time

class optStruct:
    """
    存储所有需要操作的值

    Parameters:
        dataMatIn - 数据矩阵
        classLabels - 数据标签
        C - 惩罚参数
        toler - 容错率
        kTup - 包含核函数信息的元组，第一个参数存放该核函数类别，第二个
        参数存放必要的核函数需要用到的参数

    Returns:
        None
    """

    def __init__(self, dataMatIn, classLabels, C, toler, kTup):
        # 数据矩阵
        self.X = dataMatIn
        # 数据标签
        self.labelMat = classLabels
        # 惩罚参数
        self.C = C
        # 容错率
        self.tol = toler
        # 矩阵的行数
        self.m = np.shape(dataMatIn)[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，第二
    列为实际的误差 E 的值
    self.eCache = np.mat(np.zeros((self.m, 2)))
    # 初始化核 K
    self.K = np.mat(np.zeros((self.m, self.m)))

    # 计算所有数据的核 K
    for i in range(self.m):
        self.K[:, i] = kernelTrans(self.X, self.X[i, :], kTup)

def kernelTrans(X, A, kTup):
    """
    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
    """

    # 读取 X 的行列数
    m, n = np.shape(X)

    # 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 = X * A.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):
            deltaRow = X[j, :] - A
            K[j] = deltaRow * deltaRow.T

        K = np.exp(K / (-1 * kTup[1] ** 2))

```

```
else:
    raise NameError('核函数无法识别')

return K

def loadDataSet(fileName):
    """
    读取数据

    Parameters:
        fileName - 文件名

    Returns:
        dataMat - 数据矩阵
        labelMat - 数据标签
    """

    # 数据矩阵
    dataMat = []
    # 标签向量
    labelMat = []

    # 打开文件
    fr = open(fileName)

    # 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
        lineArr = line.strip().split('\t')

        # 添加数据(100 个元素排成一行)
        dataMat.append([float(lineArr[0]), float(lineArr[1])])

        # 添加标签(100 个元素排成一行)
        labelMat.append(float(lineArr[2]))

    return dataMat, labelMat

def calcEk(oS, k):
    """
```

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

"""

```
fXk = float(np.multiply(oS.alphas, oS.labelMat).T * oS.K[:, k] + oS.b)
```

# 计算误差项

```
Ek = fXk - float(oS.labelMat[k])
```

```
return Ek
```

```
def selectJrand(i, m):
```

"""

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

"""

```
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
```

```
def selectJ(i, oS, Ei):
```

"""

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
maxK - 标号为 maxK 的数据的索引值
Ej - 标号为 j 的数据误差
"""

# 初始化
maxK = -1
maxDeltaE = 0
Ej = 0

# 根据 Ei 更新误差缓存
oS.eCache[i] = [1, Ei]

# 返回误差不为 0 的数据的索引值
validEcacheList = np.nonzero(oS.eCache[:, 0].A)[0]

# 有不为 0 的误差
if len(validEcacheList) > 1:

    # 遍历，找到最大的 Ek
    for k in validEcacheList:

        # 不计算 k==i 节省时间
        if k == i:
            continue

        # 计算 Ek
        Ek = calcEk(oS, k)

        # 计算|Ei - Ek|
        deltaE = abs(Ei - Ek)

        # 找到 maxDeltaE
        if deltaE > maxDeltaE:
            maxK = k
            maxDeltaE = deltaE
            Ej = Ek

    return maxK, Ej
```

```
# 没有不为 0 的误差
else:
    # 随机选择  $\alpha_j$  的索引值
    j = selectJrand(i, oS.m)

    # 计算  $E_j$ 
     $E_j = \text{calcEk}(\text{oS}, j)$ 

return j,  $E_j$ 

def updateEk(oS, k):
    """
    计算  $E_k$ , 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
    """

    # 计算  $E_k$ 
     $E_k = \text{calcEk}(\text{oS}, k)$ 

    # 更新误差缓存
    oS.eCache[k] = [1,  $E_k$ ]

def clipAlpha(aj, H, L):
    """
    修剪  $\alpha_j$ 

    Parameters:
        aj -  $\alpha_j$  值
        H -  $\alpha$  上限
        L -  $\alpha$  下限

    Returns:
        aj -  $\alpha_j$  值
    """
```

```
    if  $a_j > H$ :
         $a_j = H$ 

    if  $L > a_j$ :
         $a_j = L$ 

    return  $a_j$ 

def innerL(i, oS):
    """
    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
    """

    # 步骤 1: 计算误差  $E_i$ 
     $E_i = \text{calcEk}(oS, i)$ 

    # 优化  $\alpha$ , 设定一定的容错率
    if ((oS.labelMat[i] *  $E_i < -oS.tol$ ) and (oS.alphas[i] < oS.C)) or (
        (oS.labelMat[i] *  $E_i > oS.tol$ ) and (oS.alphas[i] > 0)):

        # 使用内循环启发方式 2 选择  $\alpha_j$ , 并计算  $E_j$ 
         $j, E_j = \text{selectJ}(i, oS, E_i)$ 

        # 保存更新前的  $\alpha$  值, 使用深层拷贝
         $\alpha_{old} = oS.alphas[i].copy()$ 
         $\alpha_{old} = oS.alphas[j].copy()$ 

        # 步骤 2: 计算上界 H 和下界 L
        if oS.labelMat[i] != oS.labelMat[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]

if eta >= 0:
    print("eta >= 0")

    return 0

# 步骤 4: 更新 alpha_j
oS.alphas[j] -= oS.labelMat[j] * (Ei - Ej) / eta

# 步骤 5: 修剪 alpha_j
oS.alphas[j] = clipAlpha(oS.alphas[j], H, L)

# 更新 Ej 至误差缓存
updateEk(oS, j)

if abs(oS.alphas[j] - alphaJold) < 0.00001:
    print("alpha_j 变化太小")

    return 0

# 步骤 6: 更新 alpha_i
oS.alphas[i] += oS.labelMat[i] * oS.labelMat[j] * (alphaJold - oS.alphas[j])

# 更新 Ei 至误差缓存
updateEk(oS, i)

# 步骤 7: 更新 b_1 和 b_2:
b1 = oS.b - Ei - oS.labelMat[i] * (oS.alphas[i] - alphaIold) * oS.K[i, i] -
oS.labelMat[j] * (
    oS.alphas[j] - alphaJold) * oS.K[j, i]
b2 = oS.b - Ej - oS.labelMat[i] * (oS.alphas[i] - alphaIold) * oS.K[i, j] -
oS.labelMat[j] * (
    oS.alphas[j] - alphaJold) * 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

def smoP(dataMatIn, classLabels, C, toler, maxIter, kTup=('lin', 0)):
    """
    完整的线性 SMO 算法

    Parameters:
        dataMatIn - 数据矩阵
        classLabels - 数据标签
        C - 惩罚参数
        toler - 容错率
        maxIter - 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
    """

    # 初始化数据结构
    oS = optStruct(np.mat(dataMatIn), np.mat(classLabels).transpose(), C, toler,
kTup)

    # 初始化当前迭代次数
    iter = 0
    entrieSet = True
    alphaPairsChanged = 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 < maxIter) and ((alphaPairsChanged > 0) or (entrieSet)):
        alphaPairsChanged = 0

        # 遍历整个数据集
        if entrieSet:
```

---

```

        for i in range(oS.m):
            # 使用优化的 SMO 算法
            alphaPairsChanged += 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, alphaPairsChanged))
            iter += 1

        # 遍历非边界值
        else:
            # 遍历不在边界 0 和 C 的 alpha
            nonBoundIs = np.nonzero((oS.alphas.A > 0) * (oS.alphas.A < C))[0]

            for i in nonBoundIs:
                alphaPairsChanged += 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, alphaPairsChanged))
                iter += 1

        # 遍历一次后改为非边界遍历
        if entrieSet:
            entrieSet = False

        # 如果 alpha 没有更新, 计算全样本遍历
        elif (alphaPairsChanged == 0):
            entrieSet = True
            print("迭代次数:%d" % iter)

    return oS.b, oS.alphas

def testRbf(k1=0.1):
    """
    测试函数

    Parameters:
        k1 - 使用高斯核函数的时候表示到达率

    Returns:
        None
    """

    # 开始计时
    time_start = time.time()

```

```
# 加载训练集
dataArr, labelArr = loadDataSet('testSetRBF.txt')

# 根据训练集计算 b, alphas
b, alphas = smoP(dataArr, labelArr, 200, 0.0001, 100, ('rbf', k1))

datMat = np.mat(dataArr)
labelMat = np.mat(labelArr).transpose()

# 获得支持向量
svInd = np.nonzero(alphas.A > 0)[0]
sVs = datMat[svInd]
labelSV = labelMat[svInd]
print("支持向量个数:%d" % np.shape(sVs)[0])

m, n = np.shape(datMat)

errorCount = 0

for i in range(m):
    # 计算各个点的核
    kernelEval = kernelTrans(sVs, datMat[i, :], ('rbf', k1))

    # 根据支持向量的点计算超平面，返回预测结果
    predict = kernelEval.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(labelArr[i]):
        errorCount += 1

# 打印错误率
print('训练集错误率:%.2f%%' % ((float(errorCount) / m) * 100))

# 加载测试集
dataArr, labelArr = loadDataSet('testSetRBF2.txt')

datMat = np.mat(dataArr)
labelMat = np.mat(labelArr).transpose()
m, n = np.shape(datMat)

errorCount = 0

for i in range(m):
```

```
# 计算各个点的核
kernelEval = kernelTrans(sVs, datMat[i, :], ('rbf', k1))

# 根据支持向量的点计算超平面，返回预测结果
predict = kernelEval.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(labelArr[i]):
    errorCount += 1

# 打印错误率
print('测试集错误率:%.2f%%' % ((float(errorCount) / m) * 100))

# 结束计时
time_end = time.time()
print('运行时间: ', time_end - time_start)

def showDataSet(dataMat, labelMat):
    """
    数据可视化

    Parameters:
        dataMat - 数据矩阵
        labelMat - 数据标签

    Returns:
        None
    """

    # 正样本
    data_plus = []
    # 负样本
    data_minus = []

    for i in range(len(dataMat)):
        if labelMat[i] > 0:
            data_plus.append(dataMat[i])
        else:
            data_minus.append(dataMat[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
plt.scatter(np.transpose(data_plus_np)[0], np.transpose(data_plus_np)[1])
# 负样本散点图
plt.scatter(np.transpose(data_minus_np)[0], np.transpose(data_minus_np)[1])

plt.show()

if __name__ == '__main__':
    testRbf()
```

## 附件四：使用附件三的 SMO 完整版代码求解 MNIST 问题

```
# -*- coding: utf-8 -*-
"""
处理手写数字识别问题。
此 SVM 只处理二分类问题，只取 MNIST 数据集中的 1 和 7 作二分类，其
余数字不考虑。
"""

import numpy as np
import random
import time
from os import listdir

class optStruct:
    """
    存储所有需要操作的值

    Parameters:
        dataMatIn - 数据矩阵
        classLabels - 数据标签
        C - 惩罚参数
        toler - 容错率
        kTup - 包含核函数信息的元组，第一个参数存放该核函数类别，第二个
        参数存放必要的核函数需要用到的参数

    Returns:
        None
    """

    def __init__(self, dataMatIn, classLabels, C, toler, kTup):

        # 数据矩阵
        self.X = dataMatIn
        # 数据标签
        self.labelMat = classLabels
        # 惩罚参数
        self.C = C
        # 容错率
        self.tol = toler
        # 矩阵的行数
```

```
self.m = np.shape(dataMatIn)[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，第二
列为实际的误差 E 的值
self.eCache = np.mat(np.zeros((self.m, 2)))
# 初始化核 K
self.K = np.mat(np.zeros((self.m, self.m)))

# 计算所有数据的核 K
for i in range(self.m):
    self.K[:, i] = kernelTrans(self.X, self.X[i, :], kTup)

def kernelTrans(X, A, kTup):
    """
    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
    """

    # 读取 X 的行列数
    m, n = np.shape(X)

    # 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 = X * A.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):
            deltaRow = X[j, :] - A
```

```
K[j] = deltaRow * deltaRow.T

K = np.exp(K / (-1 * kTup[1] ** 2))

else:
    raise NameError('核函数无法识别')

return K

def loadDataSet(fileName):
    """
    读取数据

    Parameters:
        fileName - 文件名

    Returns:
        dataMat - 数据矩阵
        labelMat - 数据标签
    """

    # 数据矩阵
    dataMat = []
    # 标签向量
    labelMat = []

    # 打开文件
    fr = open(fileName)

    # 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
        lineArr = line.strip().split('\t')

        # 添加数据(100 个元素排成一行)
        dataMat.append([float(lineArr[0]), float(lineArr[1])])

        # 添加标签(100 个元素排成一行)
        labelMat.append(float(lineArr[2]))

    return dataMat, labelMat
```

```
def calcEk(oS, k):
    """
    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
    """

    fXk = float(np.multiply(oS.alphas, oS.labelMat).T * oS.K[:, k] + oS.b)

    # 计算误差项
    Ek = fXk - float(oS.labelMat[k])

    return Ek


def selectJrand(i, m):
    """
    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
    """

    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


def selectJ(i, oS, Ei):
    """
```

## 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  
 maxK - 标号为 maxK 的数据的索引值  
 Ej - 标号为 j 的数据误差  
 ""

```
# 初始化
maxK = -1
maxDeltaE = 0
Ej = 0

# 根据 Ei 更新误差缓存
oS.eCache[i] = [1, Ei]

# 返回误差不为 0 的数据的索引值
validEcacheList = np.nonzero(oS.eCache[:, 0].A)[0]

# 有不为 0 的误差
if(len(validEcacheList) > 1):

    # 遍历，找到最大的 Ek
    for k in validEcacheList:

        # 不计算 k==i 节省时间
        if k == i:
            continue

        # 计算 Ek
        Ek = calcEk(oS, k)

        # 计算|Ei - Ek|
        deltaE = abs(Ei - Ek)

        # 找到 maxDeltaE
        if(deltaE > maxDeltaE):
            maxK = k
```

```
        maxDeltaE = deltaE
        Ej = Ek

    return maxK, Ej

# 没有不为 0 的误差
else:
    # 随机选择 alpha_j 的索引值
    j = selectJrand(i, oS.m)

    # 计算 Ej
    Ej = calcEk(oS, j)

    return j, Ej

def updateEk(oS, k):
    """
    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
    """

    # 计算 Ek
    Ek = calcEk(oS, k)

    # 更新误差缓存
    oS.eCache[k] = [1, Ek]

def clipAlpha(aj, H, L):
    """
    修剪 alpha_j

    Parameters:
        aj - alpha_j 值
        H - alpha 上限
        L - alpha 下限
```

---

```

Returns:
    aj - alpha_j 值
    """

    if aj > H:
        aj = H

    if L > aj:
        aj = L

    return aj

def innerL(i, oS):
    """
    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
    """

    # 步骤 1: 计算误差 Ei
    Ei = calcEk(oS, i)

    # 优化 alpha, 设定一定的容错率
    if ((oS.labelMat[i] * Ei < -oS.tol) and (oS.alphas[i] < oS.C)) or ((oS.labelMat[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 = selectJ(i, oS, Ei)

        # 保存更新前的 alpha 值, 使用深层拷贝
        alphaIold = oS.alphas[i].copy()
        alphaJold = oS.alphas[j].copy()

        # 步骤 2: 计算上界 H 和下界 L
        if oS.labelMat[i] != oS.labelMat[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]

if eta >= 0:
    print("eta >= 0")

    return 0

# 步骤 4: 更新 alpha_j
oS.alphas[j] -= oS.labelMat[j] * (Ei - Ej) / eta

# 步骤 5: 修剪 alpha_j
oS.alphas[j] = clipAlpha(oS.alphas[j], H, L)

# 更新 Ej 至误差缓存
updateEk(oS, j)

if abs(oS.alphas[j] - alphaJold) < 0.00001:
    # print("alpha_j 变化太小")

    return 0

# 步骤 6: 更新 alpha_i
oS.alphas[i] += oS.labelMat[j] * oS.labelMat[i] * (alphaJold - oS.alphas[j])

# 更新 Ei 至误差缓存
updateEk(oS, i)

# 步骤 7: 更新 b_1 和 b_2:
b1 = oS.b - Ei - oS.labelMat[i] * (oS.alphas[i] - alphaIold) * oS.K[i, i] -
oS.labelMat[j] * (oS.alphas[j] - alphaJold) * oS.K[i, j]
b2 = oS.b - Ej - oS.labelMat[i] * (oS.alphas[i] - alphaIold) * oS.K[i, j] -
oS.labelMat[j] * (oS.alphas[j] - alphaJold) * 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

def smoP(dataMatIn, classLabels, C, toler, maxIter, kTup = ('lin', 0)):
    """
    完整的线性 SMO 算法

    Parameters:
        dataMatIn - 数据矩阵
        classLabels - 数据标签
        C - 惩罚参数
        toler - 容错率
        maxIter - 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
    """

    # 初始化数据结构
    oS = optStruct(np.mat(dataMatIn), np.mat(classLabels).transpose(), C, toler,
kTup)

    # 初始化当前迭代次数
    iter = 0
    entrieSet = True
    alphaPairsChanged = 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 < maxIter) and ((alphaPairsChanged > 0) or (entrieSet)):
        alphaPairsChanged = 0

```

```
# 遍历整个数据集
if entrieSet:

    for i in range(oS.m):
        # 使用优化的 SMO 算法
        alphaPairsChanged += 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, alphaPairsChanged))
        iter += 1

# 遍历非边界值
else:
    # 遍历不在边界 0 和 C 的 alpha
    nonBoundIs = np.nonzero((oS.alphas.A > 0) * (oS.alphas.A < C))[0]

    for i in nonBoundIs:
        alphaPairsChanged += 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, alphaPairsChanged))
        iter += 1

# 遍历一次后改为非边界遍历
if entrieSet:
    entrieSet = False

# 如果 alpha 没有更新, 计算全样本遍历
elif(alphaPairsChanged == 0):
    entrieSet = True
print("迭代次数:%d" % iter)

return oS.b, oS.alphas

def img2vector(filename):
    """
    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
    """
```

```
# 创建 1*1024 零向量
returnVect = np.zeros((1, 1024))

# 打开文件
fr = open(filename)

# 按行读取
for i in range(32):
    # 读取一行数据
    lineStr = fr.readline()

    # 每一行的前 32 个数据依次存储到 returnVect 中
    for j in range(32):
        returnVect[0, 32*i+j] = int(lineStr[j])

return returnVect

def loadImages(dirName):
    """
    加载图片

    Parameters:
        dirName - 文件夹名字

    Returns:
        trainingMat - 数据矩阵
        hwLabels - 数据标签
    """

    # 训练集的 Labels
    hwLabels = []

    # 获取 trainingDigits 目录下的文件名
    trainingFileList = listdir(dirName)

    # 获取训练集数据个数
    m = len(trainingFileList)

    # 初始化训练集的 Mat 矩阵（全零阵）
    trainingMat = np.zeros((m, 1024))

    # 从文件名中解析出训练集的分类
    for i in range(m):
```

```
# 获取文件名
fileNameStr = trainingFileList[i]
fileStr = fileNameStr.split('.')[0]

# 获取文件名中表示分类的数字
classNumber = int(fileStr.split('_')[0])

# 设定 1 为负类，7 为正类
if classNumber == 1:
    hwLabels.append(-1)
elif classNumber == 7:
    hwLabels.append(1)

trainingMat[i, :] = img2vector('%s/%s' % (dirName, fileNameStr))

return trainingMat, hwLabels

def testDigits(kTup=('rbf', 50)):
    """
    测试函数

    Parameters:
        kTup - 包含核函数信息的元组

    Returns:
        None
    """

    # 训练开始计时
    time_start = time.time()

    # 加载训练集
    dataArr, labelArr = loadImages('trainingDigits')

    # 根据训练集计算 b, alphas
    b, alphas = smoP(dataArr, labelArr, 200, 0.0001, 10000, kTup)

    datMat = np.mat(dataArr)
    labelMat = np.mat(labelArr).transpose()

    # 获得支持向量
    svInd = np.nonzero(alphas.A > 0)[0]
    sVs = datMat[svInd]
```

```
labelSV = labelMat[svInd]
print("支持向量个数:%d" % np.shape(sVs)[0])

m, n = np.shape(datMat)

errorCount = 0

for i in range(m):
    # 计算各个点的核
    kernelEval = kernelTrans(sVs, datMat[i, :], kTup)

    # 根据支持向量的点计算超平面，返回预测结果
    predict = kernelEval.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(labelArr[i]):
        errorCount += 1

# 打印错误率
print('训练集错误率:%.2f%%' % ((float(errorCount) / m) * 100))

# 训练结束计时，测试开始计时
time_end_1 = time.time()
print('训练用时:', time_end_1 - time_start)

# 加载测试集
dataArr, labelArr = loadImages('testDigits')

datMat = np.mat(dataArr)
labelMat = np.mat(labelArr).transpose()

m, n = np.shape(datMat)

errorCount = 0

for i in range(m):
    # 计算各个点的核
    kernelEval = kernelTrans(sVs, datMat[i, :], kTup)

    # 根据支持向量的点计算超平面，返回预测结果
    predict = kernelEval.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(labelArr[i]):
```

```
        errorCount += 1

# 打印错误率
print('测试集错误率:%.2f%%' % ((float(errorCount) / m) * 100))

# 测试结束计时
time_end_2 = time.time()
print('测试用时:', time_end_2 - time_end_1)

if __name__ == '__main__':
    testDigits()
```

## 附件五：使用 sci-kit learn 求解 MNIST 问题

```
# -*- coding: utf-8 -*-
"""
使用 sklearn 的 SVM 处理手写数字识别问题。
此 SVM 只处理二分类问题，只取 MNIST 数据集中的 1 和 7 作二分类，其
余数字不考虑。
"""

import numpy as np
import time
from os import listdir
from sklearn.svm import SVC

def img2vector(filename):
    """
    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
    """

    # 创建 1*1024 零向量
    returnVect = np.zeros((1, 1024))

    # 打开文件
    fr = open(filename)

    # 按行读取
    for i in range(32):
        # 读取一行数据
        lineStr = fr.readline()

        # 每一行的前 32 个数据依次存储到 returnVect 中
        for j in range(32):
            returnVect[0, 32*i+j] = int(lineStr[j])

    return returnVect
```



```
def handwritingClassTest():  
    """  
    手写数字分类测试  
  
    Parameters:  
        None  
  
    Returns:  
        None  
    """  
  
    # 训练开始计时  
    time_start = time.time()  
  
    # 训练集的 Labels  
    hwLabels = []  
  
    # 返回 trainingDigits 目录下的文件名  
    trainingFileList = listdir('trainingDigits')  
  
    # 获取训练集数据个数  
    m = len(trainingFileList)  
  
    # 初始化训练集的 Mat 矩阵（全零阵）  
    trainingMat = np.zeros((m, 1024))  
  
    # 从文件名中解析出训练集的类别  
    for i in range(m):  
        # 获取文件名  
        fileNameStr = trainingFileList[i]  
  
        # 获取文件名中表示分类的数字  
        classNumber = int(fileNameStr.split('_')[0])  
  
        # 将获得的类别添加到 hwLabels  
        hwLabels.append(classNumber)  
  
        # 将每一个文件的 1*1024 数据存储到 trainingMat 矩阵中  
        trainingMat[i, :] = img2vector('trainingDigits/%s' % (fileNameStr))  
  
    # 生成 SVM 模型，指定超参数  
    clf = SVC(C=200, kernel='rbf', tol=0.0001, max_iter=10000)
```

```
# 训练模型
clf.fit(trainingMat, hwLabels)

# 训练结束计时，测试开始计时
time_end_1 = time.time()
print('训练用时:', time_end_1 - time_start)

# 返回 testDigits 目录下的文件名
testFileList = listdir('testDigits')

# 错误检测计数
errorCount = 0.0

# 获取测试集数据个数
mTest = len(testFileList)

# 从文件名中解析出测试集的分类，进行分类测试
for i in range(mTest):
    # 获取文件名
    fileNameStr = testFileList[i]

    # 获取文件名中表示分类的数字
    classNumber = int(fileNameStr.split('_')[0])

    # 将每一个文件的 1*1024 数据存储到 vectorUndertest 矩阵中
    vectorUndertest = img2vector('testDigits/%s' % (fileNameStr))

    # 获得预测结果
    classifierResult = clf.predict(vectorUndertest)

    # print("分类结果为%d\t 真实结果为%d" % (classifierResult, classNumber))

    if classifierResult != classNumber:
        errorCount += 1.0

print("误分类个数:%d\n 错误率:%f%%" % (errorCount, (errorCount / mTest)
* 100))

# 测试结束计时
time_end_2 = time.time()
print('测试用时:', time_end_2 - time_end_1)

if __name__ == '__main__':
```

handwritingClassTest()