求解SVM问题——SMO算法的推导和实现

by zwc

一、SVM原问题的描述和推导

1.问题描述

支持向量机（SVM）是一种监督学习算法，主要用于分类任务。它的基本思想是在特征空间中寻找一个超平面，使得不同类别的数据被最大间隔地分开。在最简单的形式中，线性 SVM 被用于二分类问题，即寻找一个线性超平面将两类数据分离。（针对本问题，下面仅探讨硬间隔SVM的求解）

考虑一个二元分类问题中，给定训练数据集 $(\{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\})$，其中每个 $(x_i \in \mathbb{R}^n)$是一个特征向量，$(y_i \in \{-1, 1\})$ 是对应的类别标签。支持向量机（SVM）旨在找到一个超平面：

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

这里,w是超平面的法向量， b是截距项。超平面的选择应该使得从超平面到最近的数据点的距离（即间隔）最大化。

2.优化问题推导

假设上述数据集被一个超平面分开，间隔为ρ,那么对于任意一个样本点存在以下的关系：

$$\begin{aligned} w^T x_i + b \leq -\frac{\rho}{2}, \quad \text{if } y_i = -1 \\ w^T x_i + b \geq \frac{\rho}{2}, \quad \text{if } y_i = 1 \end{aligned} \quad \Leftrightarrow \quad y_i(w^T x_i + b) \geq \frac{\rho}{2}$$

对于任意一个支持向量x_s ,上述的不等式恰好取等。在对w和b 按照$\rho/2$ 重新缩放后，每个支持向量和超平面的距离可以表示如下：

$$r = \frac{y_s(w^T x_s + b)}{||w||} = \frac{1}{||w||}$$

因此间隔ρ可以表示为$\rho = 2r = \dfrac{2}{||w||}$

最大化间隔ρ等价为最小化$||w||$最终优化问题可等价为：

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} w^T w \\ \text{subject to} \quad & y_i(w^T x_i + b) \geq 1, \quad i = 1, \ldots, n \end{aligned}$$

二、对偶问题推导

1.构建拉格朗日函数

$$L(w,b,\alpha) = 1/2 w^T w - \sum_{i=1}^{n} \alpha_i [y_i(w^T x + b) - 1]$$

其中α_i为拉格朗日乘子，非负

2.构建对偶函数

$$g(\alpha) = \inf_{w,b} \{L(w,b,a)\}$$

令拉格朗日函数L对w, b的偏导数为0：

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

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

将上述结果带入拉格朗日函数L中，可以消去w, b

对偶函数可以表示为

$$g(\alpha) = \sum_{i=1}^{n} \alpha_i - 1/2 \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^T x_j, \alpha_i \geq 0, \sum_{i=1}^{n} \alpha_i y_i = 0$$

3.构建对偶问题

通过最大化对偶函数即可得到对偶问题：

$$\text{maxmize} \quad \sum_{i=1}^{n} \alpha_i - 1/2 \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

$$\text{subject to} \quad \alpha_i \geq 0, \sum_{i=1}^{n} \alpha_i y_i = 0$$

三、KKT条件推导

1.原始约束条件

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

2.对偶约束条件

$$\alpha_i \geq 0, \forall i = 1, ..., n$$

3.互补松驰性

$$a_i[y_i(w^T x_i + b) - 1] = 0, \forall i = 1, ..., n$$

4.拉格朗日函数L对w, b的梯度为0

$$\nabla_w L = w - \sum_{i=1}^{n} \alpha_i y_i x_i = 0$$

$$\frac{\partial L}{\partial b} = -\sum_{i=1}^{n} \alpha_i y_i = 0$$

四、SMO算法

序列最小优化（SMO）算法是一种用于解决支持向量机（SVM）优化问题的算法。它是由John Platt于1998年提出的，特别适用于解决大规模的二次规划问题。SMO算法的主要优势在于其不需要昂贵的数值二次规划优化，而是将大优化问题分解为一系列最小问题来求解。

算法简要流程

1. 初始化：将所有拉格朗日乘子α_i初始化为0
2. 选择乘子：选择一对需要优化乘子α_i, α_j,这对乘子中第一个乘子选择的标准是违背KKT条件
3. 更新乘子：每次只更新选择的这对乘子，根据约束条件进行优化
4. 更新阈值b和差值e,根据新的乘子值计算新的b，和差值矩阵($w^T x + b - y$)

5. 检查收敛：若找不到需要进行优化的乘子，则说明均满足KKT条件，算法结束，否则返回步骤2继续迭代

算法关键步骤

根据KKT条件确定需要优化的乘子对中的第一个乘子

对于第一个乘子的选择称为SMO算法中的外层循环，在训练样本中选取违法KKT条件的样本点，将其对应的乘子α_i作为优化乘子对中的一个乘子

设$f(x_i) = w^T x + b$,回顾KKT条件中的对偶可行性、互补松弛性和原始约束条件

乘子α_i，及其对应的样本点需要满足以下条件：

$1.\alpha_i \geq 0$

$2.\alpha_i(y_i f(x_i) - 1) = 0$

$3.y_i f(x_i) \geq 1$

在SMO算法中常用以下方式判断乘子是否满足以上条件

```
yi = self.y[i]
alphai = self.alphas[i]
Ei = np.dot(self.w, self.X[i, :]) + self.b - self.y[i]
ri = Ei * yi
if (ri < -self.tol and alphai < self.C) or (ri > self.tol and alphai > 0):
```

注：在硬间隔问题中，设C惩罚参数无穷大即可

其中E_i指的是第i个样本的表达式$f(x_i)$和真实标签的差值

$r_i = E_i * y_i = y_i(f(x_i) - y_i) = y_i f(x_i) - (y_i)^2$

由于在本问题中的y_i的取值要么是1要么是-1，因此$r_i = y_i f(x_i) - 1$

当$r_i < 0$时（转化为程序语言即$r_i < -tolerance$),说明该样本点对应的违背了上述条件中的第3条，因此该样本点对应的乘子α_i需要进行优化

当$r_i > 0$时（转化为程序语言即为$r_i > tolerance$),说明该样本点不是支持向量，若于此同时该样本点对应的乘子$\alpha_i > 0$,则违背了上述条件中的第2条，互补松弛条件，因此该样本点对应的乘子需要进行优化

选择优化乘子对中的第二个乘子

基于最大化步长的启发式规则

确定出第一个乘子后，需要确定优化乘子对的第二个乘子。通常采用的是基于最大化步长的启发式规则，目标是选择一个使目标函数变化最大的乘子。在实践中，这通常意味着选择那个使得两个乘子对应的误差E_i和E_j之间差异最大的乘子。差异越大，可能的步长就越大，因此对目标函数的影响就越显著。采用$|E_i - E_j|$来度量这个差异

具体选择步骤

1. 首先尝试在边界样本点($\alpha_i != 0$)中寻找使得误差差值最大的乘子

```
if len(self.alphas[(self.alphas != 0) & (self.alphas != self.C)]) > 1:
   # 选择Ei矩阵中差值最大的进行优化
   # 要想|E1-E2|最大
   i = np.argmax(np.abs(self.errors - self.errors[j]))
   step_result = self.updateAlphaPair(i, j)
   if step_result:
      return 1
```

2. 若上一步中寻找的乘子不适合，则仍在边界点中随机选择第二个乘子

```
 # 循环所有非0 非C alphas值进行优化，随机选择起始点
for i in np.roll(np.where((self.alphas != 0) & (self.alphas != self.C))[0],
                 np.random.choice(np.arange(self.m))):
    step_result = self.updateAlphaPair(i, j)
    if step_result:
    return 1
```

3. 边界样本点中仍没有合适的乘子，则在全部样本点的范围进行寻找，这一步通常在算法的初始阶段进行

```
# 这里一般是程序的初始阶段
# 随机选择起始点
for i in np.roll(np.arange(self.m), np.random.choice(np.arange(self.m))):
    step_result = self.updateAlphaPair(i, j)
    if step_result:
    return 1
```

(核心)根据KKT条件更新乘子对

现在已经确定好了两个乘子α_1, α_2,进入最关键的优化环节

公式推导

根据KKT条件$\sum_{i=1}^{n}\alpha_i y_i=0$，这两个乘子之间是有约束的，下面我们通过固定除$\alpha_1$，$\alpha 2$以外的乘子，最小化$-g(\alpha)$

$$令L(\alpha_1,\alpha_2)=\frac{1}{2}[\alpha_1y_1\alpha_1y_1x_1^Tx_1+\alpha_2y_2\alpha_2y_2x_2^Tx_2+2\alpha_1y_1\alpha_2y_2x_1^Tx_2$$
$$+2\sum_{j=3}^{N}\alpha_1y_1\alpha_jy_jx_1^Tx_j+2\sum_{j=3}^{N}\alpha_2y_2\alpha_jy_jx_2^Tx_j+\sum_{j=3}^{N}\sum_{j=3}^{N}\alpha_iy_i\alpha_jy_jx_i^Tx_j$$
$$-[\alpha_1+\alpha_2+\sum_{j=3}^{N}\alpha_j]$$

$$令x_i^Tx_j=K_{ij}，以及进一步化简可得$$
$$则令L(\alpha_1,\alpha_2)=\frac{1}{2}[\alpha_1^2K_{11}+\alpha_2^2K_{22}+2\alpha_1y_1\alpha_2y_2K_{12}$$
$$+2\sum_{j=3}^{N}\alpha_1y_1\alpha_jy_jK_{1j}+2\sum_{j=3}^{N}\alpha_2y_2\alpha_jy_jK_{2j}+\sum_{j=3}^{N}\sum_{j=3}^{N}\alpha_iy_i\alpha_jy_jK_{ij}$$
$$-[\alpha_1+\alpha_2+\sum_{j=3}^{N}\alpha_j]$$

由KKT条件可知，$\sum_{i=1}^{N}\alpha_iy_i=0\Rightarrow\alpha_1y_1+\alpha_2y_2+\sum_{i=3}^{N}\alpha_iy_i=0$

不妨假设 $\sum_{i=3}^{N}\alpha_iy_i=\delta$

那么 $\alpha_1=y_1(\delta-\alpha_2y_2)$

此外，为了求导方便，在此省略了$\sum_{i=3}^{N}\sum_{j=3}^{N}\alpha_iy_i\alpha_jy_jK_{ij}+\sum_{j=3}^{N}\alpha_j$

代入α_1后，可得

$L(\alpha_2)=\frac{1}{2}[(\delta-\alpha_2y_2)^2K_{11}+2(\delta-\alpha_2y_2)\alpha_2y_2K_{12}+\alpha_2^2K_{22}$
$+2\sum_{j=3}^{N}(\delta-\alpha_2y_2)\alpha_jy_jK_{1j}+2\sum_{j=3}^{N}\alpha_2y_2\alpha_jy_jK_{2j}]-[y_1(\delta-\alpha_2y_2)+\alpha_2]$

对α_2求导

$\frac{\partial L}{\partial\alpha_2}=y_1y_2-1-\delta y_2K_{11}+\alpha_2K_{11}+\delta y_2K_{12}-2\alpha_2K_{12}$
$+\alpha_2K_{22}-\sum_{i=3}^{n}y_2\alpha_iy_iK_{1i}+\sum_{i=3}^{N}y_2\alpha_iy_jK_{2j}=0$

整理可得 $\alpha_2(K_{11}+K_{22}-2K_{12})$
$=y_2\left(y_2-y_1+\delta K_{11}-\delta K_{12}+\sum_{i=3}^{N}\alpha_iy_iK_{1i}-\sum_{i=3}^{N}\alpha_iy_iK_{2i}\right)(1)$

由KKT条件可知 $w=\sum_{i=1}^{N}\alpha_iy_ix_i$

所以 $f(x)=\sum_{i=1}^{N}\alpha_i y_i x_i^T x+b$

则 $f(x_1)-\alpha_1 y_1 K_{11}-\alpha_2 y_2 K_{12}-b=\sum_{i=3}^{N}\alpha_i y_i K_{1i}(2)$

$f(x_2)-\alpha_1 y_1 K_{12}-\alpha_2 y_2 K_{22}-b=\sum_{i=3}^{N}\alpha_i y_i K_{2i}(3)$

将式2和式3代入式1中，可得

$$\alpha_2(K_{11}+K_{22}-2K_{12})= \\ y_2(y_2-y_1+\delta K_{11}-\delta K_{12}+f(x_1)-\alpha_1 y_1 K_{11}-\alpha_2 y_2 K_{12}-b \\ -f(x_2)+\alpha_1 y_1 K_{12}+\alpha_2 y_2 K_{22}+b](4)$$

如前面所述，$\alpha_1^{old}y_1+\alpha_2^{old}y_2=\delta$,代入(4)中消去$\delta$

可得

$$\alpha_2(K_{11}+K_{22}-2K_{12}) \\ =y_2[y_2-y_1+\alpha_2^{old}y_2K_{11}+f(x_1)-2\alpha_2^{old}y_2K_{12}-f(x_2)+\alpha_2^{old}y_2K_{22}]$$

整理可得

$$\alpha_2^{new}(K_{11}+K_{22}-2K_{12}) \\ =y_2[(f(x_1)-y_1)-(f(x_2)-y_2)]+\alpha_2^{old}y_2(K_{11}+K_{22}-2K_{12})$$

令$\lambda=K_{11}+K_{22}-2K_{12}$所以 $\alpha_2^{new}=\alpha_2^{old}+\frac{y_2(E_1-E_2)}{\lambda}$

对应的更新代码如下：

```
kii = np.dot(self.X[i], self.X[i])
kjj = np.dot(self.X[j], self.X[j])
kij = np.dot(self.X[i], self.X[j])

# 计算 eta，确定乘子更新的方向和步长
eta = kii + kjj - 2 * kij

alphaJNew = alphaJOld + yJ * (EI - EJ) / eta
```

剪枝操作

由$\alpha_1+\alpha_2=\delta$可知，α_2的取值是有范围限制的，因此在上一步更新完乘子之后还需要进行剪枝操作。

假设其上限为H，下限为L

那么$\alpha_2=max(\alpha_2,L),\alpha_2=min(\alpha_2,H)$

(求解L和H,常规情况下α_1要小于等于惩罚参数C,但本问题探究硬间隔，因此当无穷处理,在实际代码中,将C设置为一个很大的常数)

假设$k=\delta$或$-\delta$

情况1： $y_1=y_2$,**则**$\alpha_1+\alpha_2=\delta(或-\delta)$

因为$C\geq\alpha_1\geq 0$,$\alpha_2=k-\alpha_1$

所有$\alpha_2\in[k-C,k]=[\alpha_1^{old}+\alpha_2^{old}-C,\alpha_1^{old}+\alpha_2^{old}]$

同时$C\geq\alpha_2\geq 0$

所以$L=max(0,\alpha_1^{old}+\alpha_2^{old}-C),H=min(C,\alpha_1^{old}+\alpha_2^{old})$

情况2： $y_1\neq y_2$,**则**$\alpha_1-\alpha_2=k$

因为$C\geq\alpha_1\geq 0,\alpha_2=\alpha_1-k$

所以$\alpha_2\in[-k,C-k]=[\alpha_2^{old}-\alpha_1^{old},C+\alpha_2^{old}-\alpha_1^{old}]$

同时$C\geq\alpha_2\geq 0$

所以$L=max(0,\alpha_2^{old}-\alpha_1^{old}),H=min(C,C+\alpha_2^{old}-\alpha_1^{old})$

对应代码如下：

```
# 计算alpha的边界
if (yI != yJ):
   # yI,yJ 异号
   L = max(0, alphaJOld - alphaIOld)
   H = min(self.C, self.C + alphaJOld - alphaIOld)
elif (yI == yJ):
   # y1,y2
   L = max(0, alphaIOld + alphaJOld - self.C)
   H = min(self.C, alphaIOld + alphaJOld)
```

更新截距b和差值向量e

更新完乘子之后，还需要进一步更新截距和差值向量

当$\alpha_1^{new}\in(0,C)$时，有$y_if(x_i)=1$，又根据KKT条件有$w=\sum\alpha_iy_ix_i$

可以得到$\sum_{i=1}^{N}\alpha_iy_iK_{i1}+b=y_1(1)$

所以有$b_1^{new}=y_1-\sum_{i=3}^{N}\alpha_i y_i k_{i1}-\alpha_1^{new}y_1K_{11}-\alpha_2^{new}y_2K_{21}$

$E_1=f(x_1)-y_1=\sum_{i=3}^{N}\alpha_i y_i K_{i1}+\alpha_1^{old}y_1K_{11}+\alpha_2^{old}y_2K_{21}+b^{old}-y_1$

即$y_1-\sum_{i=3}^{N}\alpha_i y_i K_{i1}=-E_1+\alpha_1^{old}y_1K_{11}+\alpha_2^{old}y_2K_{21}+b^{old}(2)$

将式2代入式1，可得

$b_1^{new}=-E_1-y_1K_{11}(\alpha_1^{new}-\alpha_1^{old})-y_2K_{21}(\alpha_2^{new}-\alpha_2^{old})+b^{old}$

同理

$b_2^{new}=-E_2-y_1K_{12}(\alpha_1^{new}-\alpha_1^{old})-y_2K_{22}(\alpha_2^{new}-\alpha_2^{old})+b^{old}$

如果α_1^{new}或$\alpha_2^{new}\in(0,C)$,那么$b^{new}=b_1^{new}$或b_2^{new}

如果两者都是0或者C,则$b^{new}=(b_1^{new}+b_2^{new})/2$

更新完b之后，再重新计算差值向量即可。

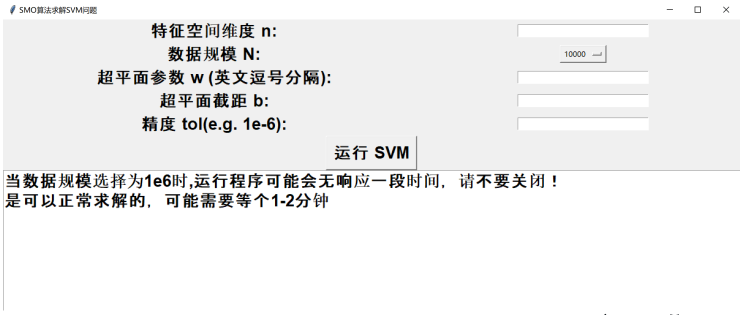
对应代码如下：

```
# 计算新截距b的值
b1 = self.b - (EI + yI * (alphaINew - alphaIOld) * kii
                + yJ * (alphaJNew - alphaJOld) * kij)
b2 = self.b - (EJ + yI * (alphaINew - alphaIOld) * kij
                + yJ * (alphaJNew - alphaJOld) * kjj)
# 更新截距
if 0 < alphaINew:
     b_new = b1
elif 0 < alphaJNew:
     b_new = b2
else:
     b_new = (b1 + b2) / 2

# 更新w的值
self.w = self.w + yI * (alphaINew - alphaIOld) * self.X[i]
        + yJ * (alphaJNew - alphaJOld) * self.X[j]

# 同时更新差值矩阵其它值
self.errors = self.allE()

# 计算所有样本的误差值
def allE(self):
    return np.dot(self.X, self.w) + self.b - self.y
```

五、实验

实验内容

使用Tkinter创建了简单的用户交互界面：



需要用户输入特征空间的维度n，选择数据的规模$N \in \{10^4, 10^5, 10^6\}$,输入与空间维度匹配的超平面参数$w, b$以及精度$tol$

点击运行SVM后，程序会先根据输入的参数生成数据集，生成数据的代码如下：

```
def generate_balanced_data(n, N, w, b):

    X = np.random.randn(int(N),int(n))
    y = np.zeros(N)
    np.random.seed(125)
    norm_w = np.linalg.norm(w)
    range_min = -norm_w * (1+abs(b))
    range_max = norm_w * (1+abs(b))
    # Half data points with label +1
    for i in range(N // 2):
        while True:
            point = np.random.uniform(range_min,range_max,n)
            if np.dot(point, w) + b > range_max:
                X[i] = point
                y[i] = 1
                break
    # Half data points with label -1
    for i in range(N // 2,N):
        while True:
            point = np.random.uniform(range_min,range_max,n)
            if np.dot(point, w) + b <-range_max:
                X[i] = point
                y[i] = -1
                break
    return X, y
```

生成数据后，程序将在这些数据上运行SMO算法，求解最优分隔超平面并记录时间开销。

```
# 创建并训练 SVM 模型
    svm = SVM(X, y, tol)
    start = time.time()
    svm.fit()
    end = time.time()
```

求解出来的超平面，以及求解过程中得到的拉格朗日乘子将再次用来完整地验证4个KKT条件。检验KKT条件的片段代码如下:

```
# 1. Original constraints: y_i(w^T x_i + b) >= 1 for all data points
original_constraints_satisfied =
[y_i * (np.dot(w, x_i) + b)>=1-tol for x_i, y_i in zip(X, y)]
proportion_satisfied = sum(original_constraints_satisfied)/len(y)
```

```
# 2. Lagrange multipliers: lambda_i >= 0 for all lambda
lambda_positive_satisfied = all(lambda_i >= 0-tol for lambda_i in alpha)
```

```
# 3. Complementary slackness: lambda_i * (1 - y_i(w^T x_i + b)) = 0
comp_slack_sat = [abs(lambda_i * (1 - y_i * (np.dot(w, x_i) + b)))< tol
                            for x_i, y_i, lambda_i in zip(X, y, alpha)]
comp_slack_sat = sum(comp_slack_sat)/len(comp_slack_sat)
```

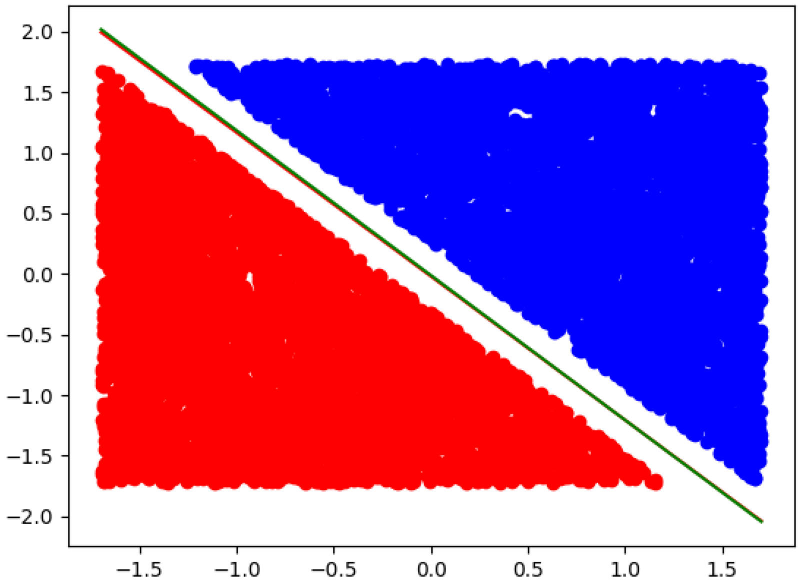
```
# 4. Gradient condition:
# norm(w - sum(lambda_i * y_i * x_i)) < 1e-6
# sum(lambda_i*y_i)=0
grad_cond_w_sat = np.linalg.norm(w - sum(lambda_i * y_i * x_i
                    for x_i, y_i, lambda_i in zip(X, y, alpha))) < tol
grad_cond_b_sat = sum([lambda_i * y_i
                            for y_i, lambda_i in zip(y, alpha)]) < tol
grad_cond_sat = grad_cond_w_sat&grad_cond_b_sat
```

结果

以下呈现部分实验结果，程序可在较短的时间内获得较为理想的结果:

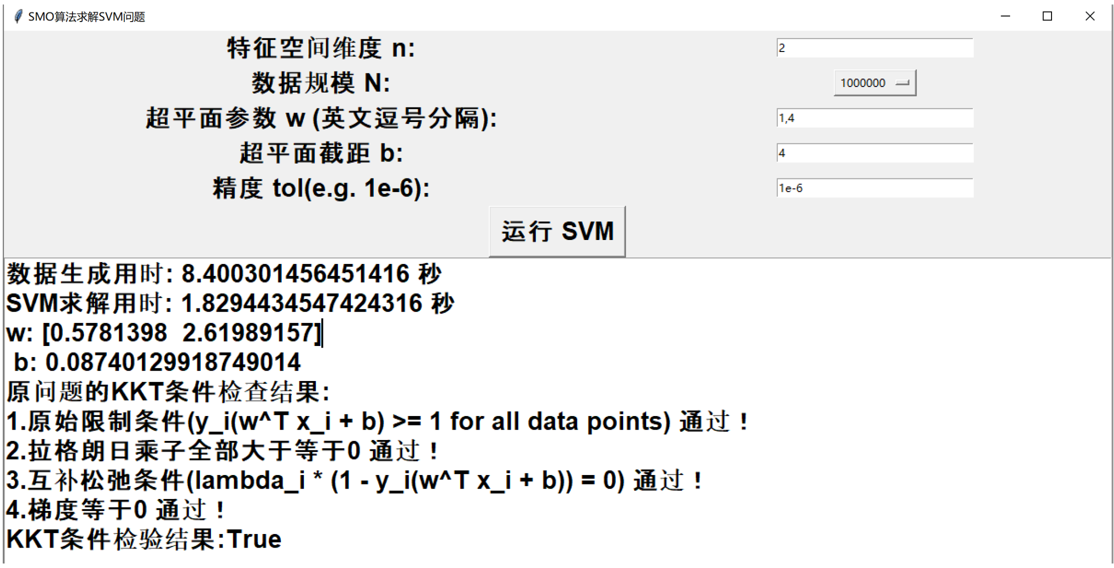
2维测试——绘图与Sklearn库的结果进行对比

由于3种规模的绘图结果基本一致，此处只展示一万规模的数据点的图像。本程序求解的超平面和Sklearn求解所得基本完全重合，中间的红色超平面为Sklearn求解结果，绿色超平面为本程序求解结果:



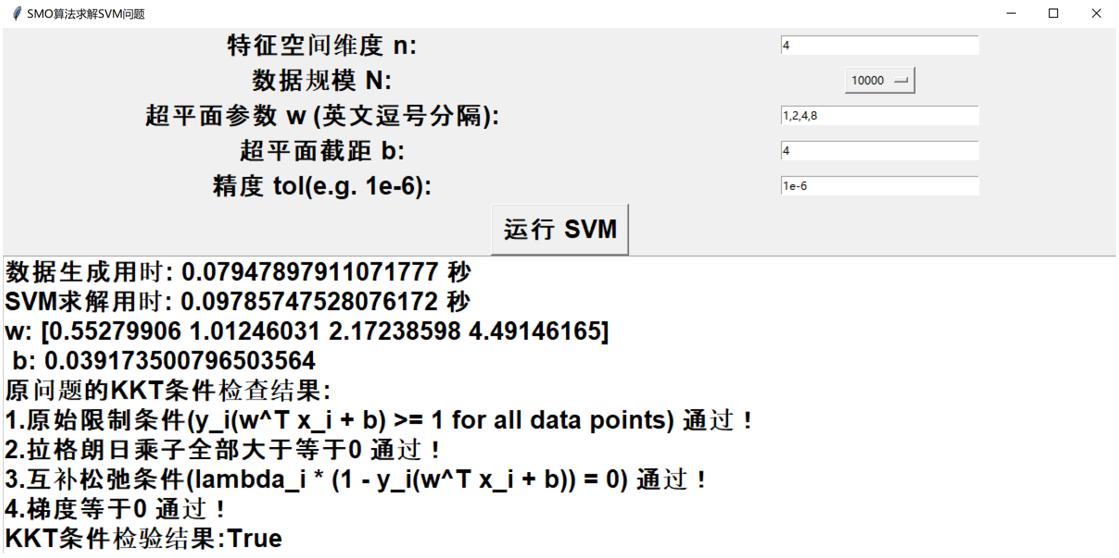
（大多数情况下，红线和绿线基本重合，上图是为了描述有红绿线的区别特地找的一组有点不重合的例子）

求解一百万的数据规模，（仅针对2维数据）时间开销也仅需1.8s

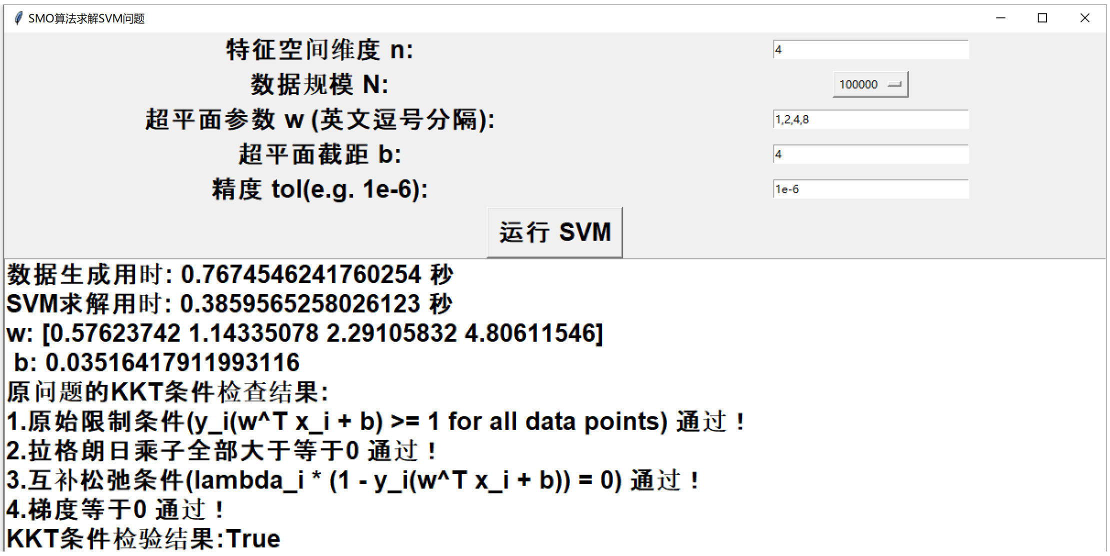


4维测试

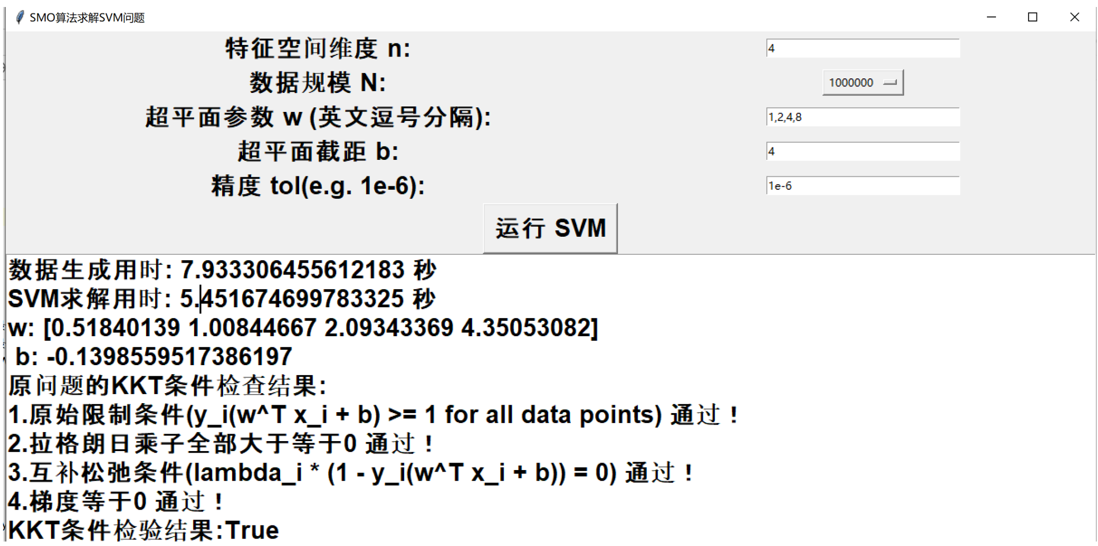
规模10^4



规模10^5



规模10^6



实验结果分析

数据量达到一百万时仍可以在较短时间内计算完毕

(不同设备运行的速度可能有所不同，增加维度也将显著加大时间开销)

由于生成数据的逻辑如下：

```
for i in range(N // 2):
    while True:
        point = np.random.uniform(range_min,range_max,n)
        # Ensure the point is above the hyperplane
        if np.dot(point, w) + b > 1(or < -1):
            X[i] = point
            y[i] = 1
            break
```

因此最佳超平面应该接近生成数据时预设的超平面，从实验结果来看，最重要的是均可以在较短时间内解得满足KKT条件验证的超平面参数；其次是，求解得到的w*和预设的w基本上成标量倍，说明求解所得超平面平行于预设超平面，这是符合实际的。

此外，由于预设的超平面未必就是最优超平面，因此截距b未必成相同的标量倍。

Reference

[1] "详细剖析SMO算法中的知识点" - 知乎专栏. 发布于2021年12月5号 网址：https://zhuanlan.zhihu.com/p/433150785

[2] "机器学习之利用SMO算法求解支持向量机—基于python" - CSDN博客. 发布于2023年4月20号 网址：https://blog.csdn.net/qq_45856698/article/details/130250794

[3] J. Platt, “Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines,” Technical Report MSR-TR-98-14, Microsoft Research, 1998.