机器学习基础 实验五 实验报告

学号：202122202214

姓名：马贵亮

班级：2021级软件5班

实验目的

掌握线性SVM的基本原理和基本设计步骤；

掌握非线性SVM的基本原理和基本设计步骤。

实验内容

- 使用exp5_1.mat数据，构造线性SVM对数据进行划分，给出可视化的划分边界结果。
- 探究不同程度的惩罚因子C对样本分类误差的影响。
- （选做）构造使用Gaussian kernels的SVM, 对exp5_2.mat数据进行划分，给出可视化的划分边界的结果。

- 编程语言不限，推荐使用python；
- xxx.mat数据文件可以使用scipy.io进行读取处理，模型需要自己实现，不可使用已有的第三方库；
- 实验报告中提供的代码需要有必要的注释。

实验过程

1. SVM与惩罚因子相关知识

1.1 VC维与SVM

从VC维（Vapnik-Chervonenkis维度）的角度来看，支持向量机（SVM）最大化间隔（margin）是一种有效的方式来控制模型的复杂度和提升泛化性能。VC维是衡量模型复杂度的一种指标，它表示一个模型在有限样本下能够被正确分类的最大样本数量。

VC维越高，模型的复杂度越高，意味着模型能够拟合更多的样本。高VC维的模型虽然在训练集上可能表现良好，但在面对未见过的数据时容易出现过拟合，泛化能力较差。控制VC维可以帮助平衡模型的复杂度和泛化能力。通过减少VC维，可以防止过拟合，提升模型在新数据上的表现。

在SVM中，最大化间隔相当于在高维特征空间中找到最优的分离超平面，使得两个类别之间的最小距离最大化。研究表明，间隔越大，VC维越低，这意味着模型的复杂度降低，从而减少了过拟合的风险。这是因为较大的间隔能够让模型对训练数据中的微小噪声和变动更具鲁棒性。

较大的间隔使得分类决策面远离训练样本点，这样即使在测试集中出现一些偏离训练集分布的样本点，模型也能更好地进行分类。换句话说，最大化间隔增强了模型的鲁棒性和容错能力。较低的VC维意味着模型具有更好的泛化性能，因此SVM通过最大化间隔来控制VC维，最终实现提升泛化能力的目的。

Vapnik等人提出的理论表明，SVM通过最大化间隔来最小化结构风险（Structural Risk Minimization, SRM），这是一种比仅仅最小化经验风险（Empirical Risk Minimization, ERM）更为有效的学习策略。SRM考虑了模型复杂度（VC维）和训练误差，从而更好地控制泛化误差。

1.2 SVM基本表示形式

SVM线性平面方程可以描述为：$\mathbf{w}^T\mathbf{x}+b=0$，假设间隔为$d$

通过一些列等价变化可以将分类的边界变为：

$$\begin{cases}\mathbf{w}^T\mathbf{x}+b\geqslant+1,\ y_i=+1\\ \mathbf{w}^T\mathbf{x}+b\leqslant-1,\ y_i=-1\end{cases}$$

通过点到超平面的距离公式可以得知

$$r=\frac{|\mathbf{w}^T+b|}{||\mathbf{w}||}$$

$$d=\frac{|1|}{||\mathbf{w}||}+\frac{|-1|}{||\mathbf{w}||}=\frac{2}{||\mathbf{w}||}$$

SVM的过程即最大化 间隔d，即：

$$\max_{\mathbf{w},b}\quad\frac{2}{||\mathbf{w}||}$$

$$s.t.\quad y_i(\mathbf{w}^T\mathbf{x}_i+b)-1\geqslant0\quad i=1,2,\ldots,m$$

该问题等价于：

$$\min_{\mathbf{w},b}\quad\frac{1}{2}||\mathbf{w}||^2$$

$$s.t.\quad y_i(\mathbf{w}^T\mathbf{x}_i+b)-1\geqslant0\quad i=1,2,\ldots,m$$

1.3 核函数

在实际应用中，许多数据集不是线性可分的，这意味着没有一个直线或平面能够完全正确地分类数据点。为了解决这个问题，SVM使用了一种叫做核技巧的方法，允许SVM在更高维的空间中有效地进行工作，而无需直接计算高维空间中的点。

核函数可以将输入数据映射到一个高维空间，其中数据点更有可能是线性可分的。核函数的选择取决于数据的特性和问题类型。常见的核函数包括：

- **线性核（Linear Kernel）**：没有进行任何映射，保持数据在原始空间中不变，适用于线性可分的数据集。
- **多项式核（Polynomial Kernel）**：将数据映射到一个多项式特征空间，适用于非线性问题。
- **径向基函数核（RBF，也称为高斯核）**：一种非常流行的核，可以映射到无限维的空间，非常适合处理那些在原始空间中呈复杂分布的数据。
- **Sigmoid核**：类似于神经网络中使用的激活函数。

核函数 $K(\mathbf{x},\mathbf{y})$ 是一个函数核函数，对于所有 $\mathbf{x}$ 和 $\mathbf{y}$ 在输入空间中，它返回这两个点在高维特征空间中的内积，即：

$$K(\mathbf{x},\mathbf{y})=<\phi(\mathbf{x}),\phi(\mathbf{y})>$$

这里的$<,>$表示内积，而 ϕ 表示从输入空间到某个高维特征空间的映射。

常见的核函数：

线性核：

$$K(\mathbf{x},\mathbf{y})=\mathbf{x}^T\mathbf{y}$$

多项式核：

$$K(\mathbf{x},\mathbf{y})=(\mathbf{x}^T\mathbf{y}+c)^d$$

高斯核（RBF）：

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

拉普拉斯核：

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

Sigmoid核：

$$K(\mathbf{x},\mathbf{y})=tanh(\alpha\mathbf{x}^T\mathbf{y}+c)$$

在实际应用时，只需要将$D_{ij}=y_iy_j\mathbf{x_i^T}\mathbf{x_j}=(\mathbf{y}\mathbf{y^T})\odot(XX^T)$修改为$D_{ij}=y_iy_j\mathbf{x_i^T}\mathbf{x_j}=(\mathbf{y}\mathbf{y^T})\odot K(\mathbf{x}_i,\mathbf{x}_j)$即可

1.4 SVM对偶形式（强对偶）

基于凸规划的表示形式，可以采用拉格朗日乘子法来对原问题进行求解

原问题：

$$\begin{aligned}\min_{\mathbf{w},b}\quad&\frac{1}{2}||\mathbf{w}||^2\\ s.t.\quad&y_i(\mathbf{w}^T\mathbf{x}_i+b)-1\geqslant 0\quad i=1,2,\dots,m\end{aligned}$$

引入拉格朗日因子λ后形式如下：

$$L(\mathbf{w},b;\lambda)=\frac{1}{2}||\mathbf{w}||^2+\sum_{i=1}^m\lambda_i(1-y_i(\mathbf{w}^T\mathbf{x}_i+b))$$

$$\nabla_{\mathbf{w},b}L(\mathbf{w},b;\lambda)=\mathbf{0}$$

$$\nabla_{\mathbf{w}}L(\mathbf{w},b;\lambda)=\mathbf{w}-\sum_{i=1}^m\lambda_iy_i\mathbf{x}_i=\mathbf{0}\quad\Rightarrow\quad\mathbf{w}^*=\sum_{i=1}^m\lambda_iy_i\mathbf{x}_i$$

$$\nabla_bL(\mathbf{w},b;\lambda)=\sum_{i=1}^m\lambda_iy_i=0$$

带入即可得到其拉格朗日对偶问题：

$$\begin{aligned}\max_{\mathbf{\Lambda}}\quad&L(\mathbf{w}*,b*;\mathbf{\Lambda})\\ s.t.\quad&\nabla_{\mathbf{w},b}L(\mathbf{w},b;\mathbf{\Lambda})=\mathbf{0}\\&\lambda_i\geq 0,\quad i=1,\dots,m\end{aligned}$$

带入展开后可得：

$$\begin{aligned}\max_{\mathbf{\Lambda}}\quad&\sum_{i=1}^m\lambda_i-\frac{1}{2}\sum_{i=1}^m\sum_{j=1}^m\lambda_i\lambda_jy_iy_j\mathbf{x}_i^T\mathbf{x}_j\\ s.t.\quad&\textstyle\sum_{i=1}^m\lambda_iy_i=0\\&\lambda_i\geq 0,\quad i=1,\dots,m\end{aligned}$$

引入矩阵形式则为：

$$\max_{\mathbf{\Lambda}} \quad \mathbf{1}^T\mathbf{\Lambda} - \frac{1}{2}\mathbf{\Lambda}^T D\mathbf{\Lambda}$$
$$s.t. \quad \begin{aligned} y^T\mathbf{\Lambda} &= \mathbf{0} \\ \mathbf{\Lambda} &\geq 0 \end{aligned}$$

其中 $D_{ij} = y_i y_j \mathbf{x_i^T x_j} = (\mathbf{y y^T}) \odot (XX^T)$

该问题为一个QR问题，可以采用内点法或者SMO进行求解，由于原问题是由拉格朗日乘数变化而来，因此在强对偶的前提下一定满足KKT定理，因此对于那些在 $\mathbf{w} \cdot \mathbf{x}_i + b = \pm 1$ 上的点对应的λ_i不为0， 其余的λ_i 为0，那些在线上的点称之为支持向量，因此对于一个模型我们可以仅保存支持向量即可。

由于后续会存在利用核函数进行升维度的操作，因此$\mathbf{w}$ 的维度并不好确定，但是根据拉普拉斯函数的约束条件我们可以得知：

$$\mathbf{w}^* = \sum_{i=1}^{m} \lambda_i y_i \mathbf{x}_i$$

如果我们将支持向量进行存储，那么首先可以根据支持向量之间的关系求解 b，设支持向量的集合为 X_{sv}, y_{sv}，则：

$$b = avg(y_{sv} - X_{sv}\mathbf{w}) = avg(y_{sv} - X_{sv}\sum_{i=1}^{sv} \lambda_i y_{svi} \mathbf{x_{sv}}_i)$$

那么对于预测即为，在此处将支持向量集合写作 X, y

$$y_{pred} = \sum_{i=1}^{m} \lambda_i y_i \mathbf{x}_i^T \mathbf{x} + b$$

引入核函数则为：

$$y_{pred} = \sum_{i=1}^{m} \lambda_i y_i K(\mathbf{x}_i, \mathbf{x}) + b$$

1.5 软间隔相关

在实际应用中，数据往往不是完全线性可分的，这就需要使用软间隔（soft margin）的概念来处理轻微的数据重叠。

松弛变量（Slack Variables）

对于每一个训练样本，引入一个松弛变量ξ_i。这个变量表示第i个数据点违反边界的程度。如果，则$\xi_i = 0$该点正确分类并且在边界外。如果 ，则该$\xi_i > 0$点在边界内或者被错误分类。

那么原问题要优化的函数则变为：

$$\min_{\mathbf{w}, b, \xi_i} \quad \frac{1}{2}||w||^2 + C\sum_{i=1}^{m} \xi_i$$
$$s.t. \quad \begin{aligned} y_i(\mathbf{w}^T\mathbf{x}_i + b) &\geq 1 - \xi_i \\ \xi_i &\geq 0 \quad i = 1, 2, \ldots, m \end{aligned}$$

再次引入拉格朗日乘子：

$$L(\mathbf{w}, b, \mathbf{\Xi}; \mathbf{\Lambda}, \mathrm{M}) = \frac{1}{2}||w||^2 + C\sum_{i=1}^{m} \xi_i + \sum_{i=1}^{m} \lambda_i (1 - \xi_i - y_i(\mathbf{w}^T\mathbf{x}_i + b)) - \sum_{i=1}^{m} \mu_i \xi_i$$

其中 $\lambda_i \geq 0, \mu_i \geq 0$ 均为拉格朗日乘子，令$L(\mathbf{w}, b, \boldsymbol{\Xi}; \boldsymbol{\Lambda}, \mathrm{M})$，对$\mathbf{w}, b, \boldsymbol{\Xi}$ 求偏导可得：

$$\mathbf{w} = \sum_{i=1}^{m} \lambda_i y_i \mathbf{x}_i$$

$$0 = \sum_{i=1}^{m} \lambda_i y_i$$

$$C = \lambda_i + \mu_i$$

带入后可以得到其对偶问题

$$\underset{\Lambda}{max} \quad \sum_{i=1}^{m} \lambda_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$$

$$s.t. \quad \sum_{i=1}^{m} \lambda_i y_i = 0$$

$$0 \leq \lambda_i \leq C, \quad i = 1, 2, \ldots, m$$

则应满足KKT 条件即：

$$\begin{cases} \lambda_i \geq 0, \\ \mu_i \geq 0, \\ y_i f(\mathbf{x}_i) - 1 + \xi_i \geq 0, \\ \lambda_i (y_i f(\mathbf{x}_i) - 1 + \xi_i) = 0, \\ \xi_i \geq 0, \\ \mu_i \xi_i = 0. \end{cases}$$

整体优化表达式与无软间隔的表达式除了在引入的惩罚因子和拉格朗日乘子μ_i 存在约束外，最本质的在于对于λ_i 的约束存在了一个上限C

1.6 支持向量

在支持向量机（SVM）中，**支持向量**（Support Vectors）是指那些位于决策边界附近并直接影响到分类决策的训练样本。具体来说，支持向量具有以下几个特点：

1. **边界样本**：
    - 支持向量是距离分类决策面（或称超平面）最近的训练样本点。它们是定义最大间隔的关键点。
    - 这些样本点位于分类间隔（margin）边缘，即位于到决策面有相同最小距离的两条平行超平面上。
2. **影响决策边界**：
    - 决策边界（超平面）是由这些支持向量确定的。移动或改变支持向量会导致决策边界的位置和方向发生变化。
    - 其他训练样本（不在支持向量集合中的样本）对决策边界没有直接影响，只要它们在支持向量定义的间隔之外。
3. **数学定义**：
    - 在线性SVM中，决策函数可以表示为 $f(x) = \mathbf{w}^T \mathbf{x} + b$，其中 $\mathbf{w}$是权重向量，b 是偏置。
    - 支持向量是满足以下约束的样本点：$y_i(\mathbf{w}^T \mathbf{x}_i + b) = 1$，其中 y_i 是样本 $\mathbf{x}_i$ 的标签（+1 或 -1）。
    - 在非线性SVM中，通过核方法将数据映射到高维特征空间后，支持向量依然是那些距离决策超平面最近的点。
4. **稀疏性**：

- 在训练SVM时，并非所有样本点都成为支持向量。通常，只有一部分样本点（支持向量）对模型的最终决策面有贡献。
- 这种稀疏性使得SVM在处理大规模数据时具有一定的效率优势，因为决策函数的计算主要依赖于支持向量。

2. 基于梯度下降和合页损失的 SVM

梯度下降是一种优化算法，用于通过不断调整参数来最小化损失函数。以下是梯度下降过程的详细解释：

2.1 初始化参数

首先，算法需要初始化参数（在我们的 SVM 实现中是权重 $\mathbf{w}$ 和偏置 b）。这些参数可以随机初始化或设置为零。

2.2 计算损失函数

损失函数衡量模型预测与实际结果之间的差异。在支持向量机中，常用的损失函数是合页损失函数（hinge loss），加上一个正则化项来防止过拟合。

合页损失函数的形式如下：

$$L = C \sum_{i=1}^{n} \max(0, 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b)) + \frac{1}{2}||\mathbf{w}||^2$$

其中，y_i 是第 i 个样本的标签，$\mathbf{x}_i$ 是第 i 个样本的特征向量，λ 是正则化参数。

2.3 计算梯度

梯度是损失函数相对于参数的导数，表示损失函数在当前参数值下变化的方向和速度。在我们的 SVM 实现中，梯度的计算如下：

- 当样本满足 $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$ 时，损失函数对参数的梯度为：

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w}$$
$$\frac{\partial L}{\partial b} = 0$$

- 当样本不满足 $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$ 时，损失函数对参数的梯度为：

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - Cy_i\mathbf{x}_i$$
$$\frac{\partial L}{\partial b} = -Cy_i$$

2.4 更新参数

利用梯度更新参数。更新的规则是沿着梯度的反方向调整参数，因为梯度指向的是损失函数增加最快的方向，而我们希望减少损失。

更新公式如下：

$$\mathbf{w} = \mathbf{w} - \eta \frac{\partial L}{\partial \mathbf{w}}$$
$$b = b - \eta \frac{\partial L}{\partial b}$$

其中，η 是学习率，决定了每次更新的步长。

2.5 重复迭代

重复步骤 2 到 4，直到达到预定的迭代次数或者损失函数的变化趋于平缓，即收敛。

2.6 代码实现

整体代码实现难度不大，和之前线性分类器的时间基本保持一致，只不过在进行梯度更新的过程中需要对当前满足的条件进行判定，以此来选择不同的偏导。关键在于软硬间隔的实现上，由于这种方式的C是一个约束在合页损失函数上的，如果C=0，那么合页损失项完全损失，只剩下正则化的部分，此时C=0和不存在C并不相同。

但是，在我对数据进行测试过程中，当我将软间隔设定的非常大时其效果与硬间隔保持一致。

在返回后返回支持向量，并且将系数存储在对应的类中。

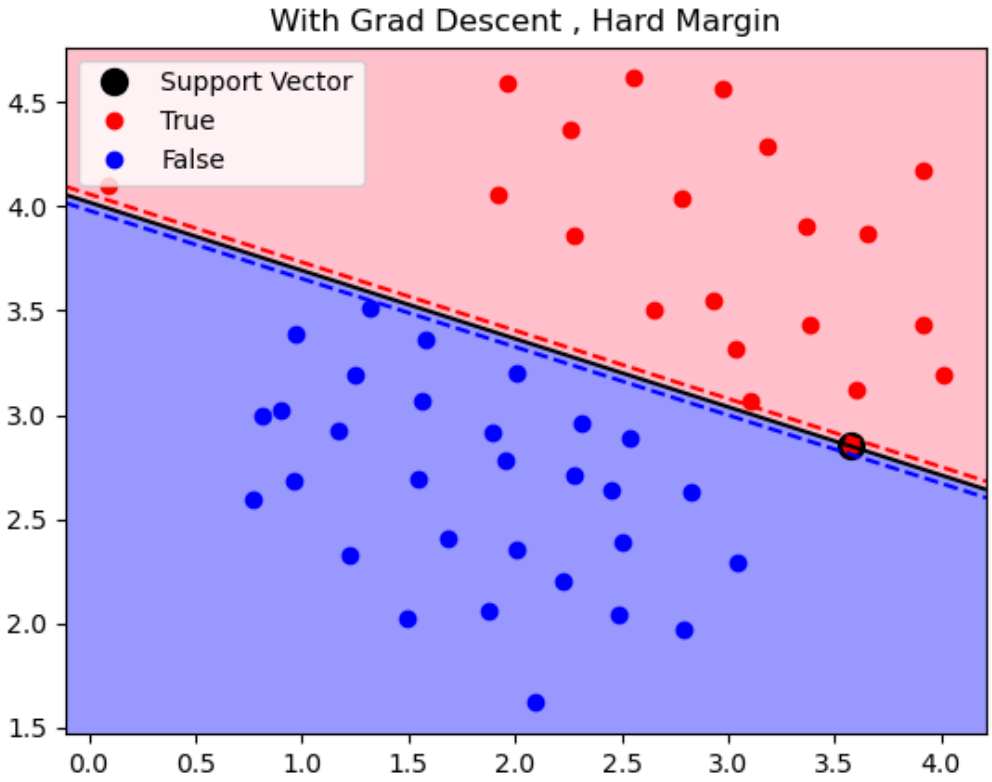
```
    def fitByGradDescent(self, X, label, margin='soft', iters=10000, learning_rate=0.00001, kernel='none'):
        self.kernel = kernel
        self.trained = True
        X = X.values
        X = np.array(X)
        n_samples, n_features = X.shape
        y = label.astype(float)
        y[label == 0] = -1
        y = np.array(y).reshape(-1, 1)
        if margin == 'hard':
            self.C = 10000

        if kernel != 'none':
            raise ValueError("Grad Descent only support kernel 'none'")

        self.weight = np.zeros(n_features)
        self.bias = 0
        for _ in range(iters):
            for index, x_i in enumerate(X):
                condition = (y[index] * (np.dot(x_i, self.weight.reshape(-1, 1)) + self.bias)) >= 1
                if condition:
                    self.weight -= learning_rate * (self.weight)
                else:
                    self.weight -= learning_rate * (self.weight - self.C * x_i * y[index])
                    self.bias -= -learning_rate * self.C * y[index]

        self.stategy = 'grad'
        sv_X = []
        sv_y = []
        self.weight = self.weight.reshape(-1, 1)
        for index, X_i in enumerate(X):
            if y[index] * (np.dot(X_i, self.weight) + self.bias) - 1 < 1e-2:
                sv_X.append(X_i)
                sv_y.append(y[index])
        self.sv_X = np.array(sv_X)
        self.sv_y = np.array(sv_y)
        return self.sv_X

    def predict(self, X):
        if self.stategy == "grad":
```

```
            y = np.dot(X, self.weight) + self.bias
            return y
```

随后根据对应的支持向量和 `predict` 函数来进行绘图，其他的部分的绘图的过程和本阶段保持一致，并且在绘图中绘制对应的支持向量。

```
    sv_X #为返回的支持向量

    x_min, x_max = X['x1'].min(), X['x1'].max()
    y_min, y_max = X['x2'].min(), X['x2'].max()
    x_thresold = (x_max - x_min) / 20
    y_thresold = (y_max - y_min) / 20
    x_min -= x_thresold
    x_max += x_thresold
    y_min -= y_thresold
    y_max += y_thresold
    custom_cmap = ListedColormap(['#0000ff',  # 浅红
                                  '#000000',
                                  '#ff0000'])

    custom_cmap2 = ListedColormap(['#9898ff',  # 浅红
                                   '#FFC0CB', ])

    plt.plot(sv_X[:, 0], sv_X[:, 1], 'k.', markersize=20, label='Support
Vector')
    plt.plot(X_true['x1'], X_true['x2'], 'r.', markersize=12, label='True')
    plt.plot(X_false['x1'], X_false['x2'], 'b.', markersize=12,
label='False')
    x0, x1 = np.meshgrid(np.linspace(x_min, x_max, 1000).reshape(-1, 1),
                         np.linspace(y_min, y_max, 1000).reshape(-1, 1))
    X_new = np.c_[x0.ravel(), x1.ravel()]
    Y_new = model.predict(X_new)
    z = Y_new.reshape(x0.shape)
    z_label = np.where(z > 0, 1, -1)

    plt.contourf(x0, x1, z_label, cmap=custom_cmap2)
    plt.contour(x0, x1, z, levels=[-1, 0, 1], linestyles=['--', '-', '--'],
cmap=custom_cmap)
    plt.axis([x_min, x_max, y_min, y_max])
    plt.show()
```

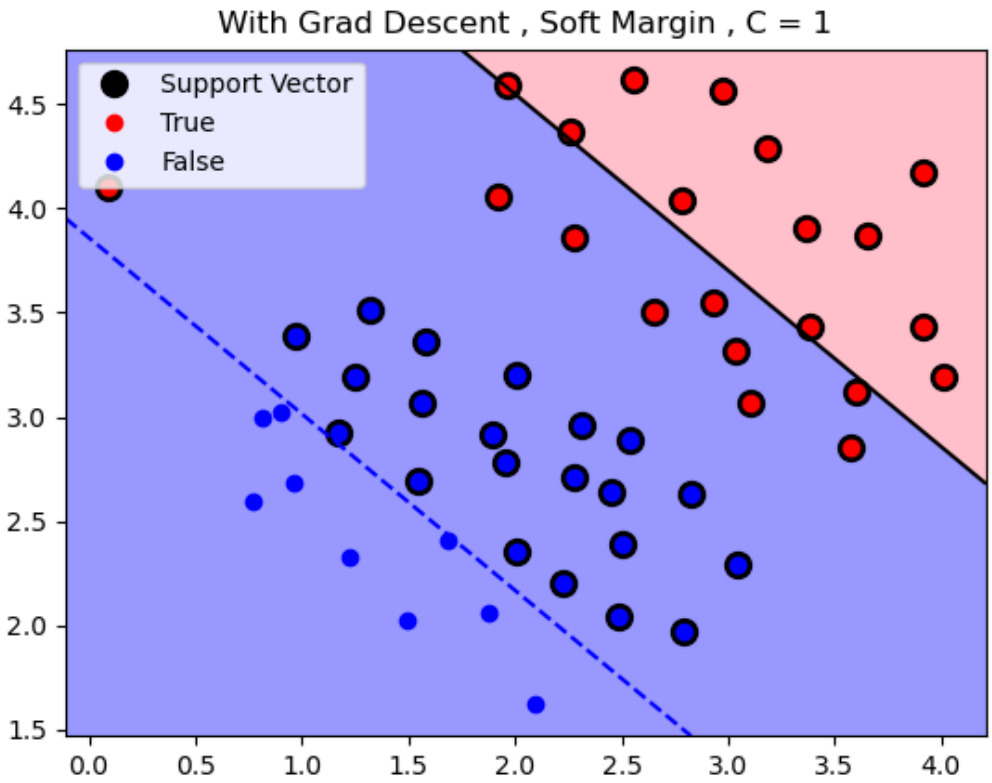
当我采用硬间隔时，得到如下结果：



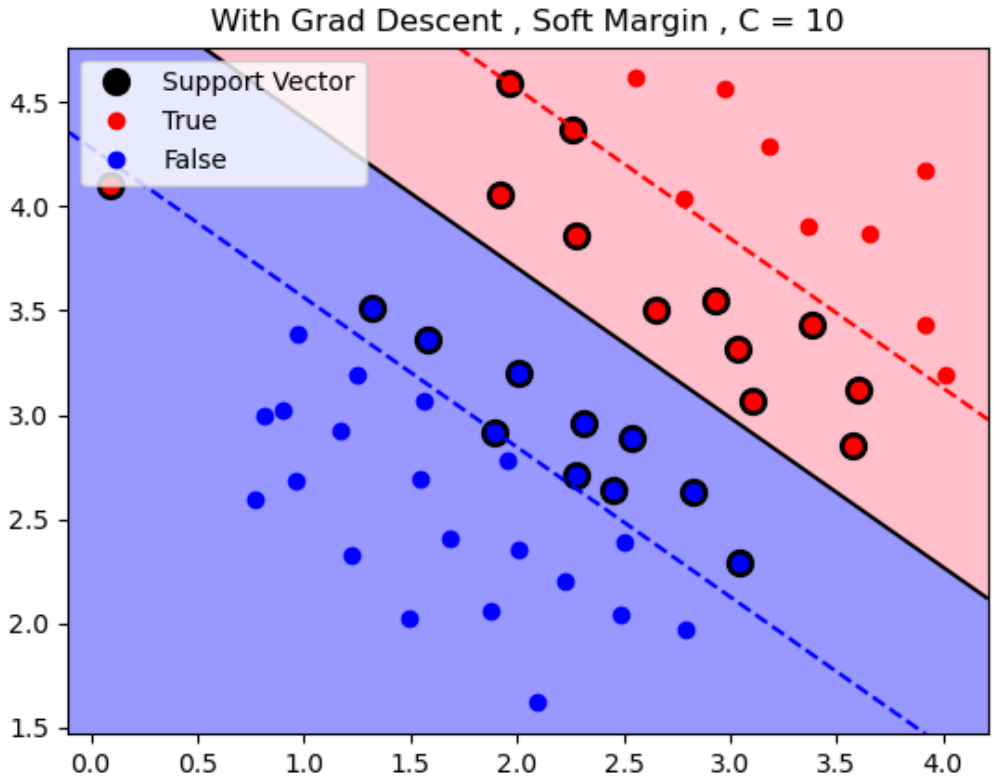
由于是通过设置一个非常大的惩罚因子所生成，因此本质其只是非常接近硬间隔，而非本质的硬间隔，真正的硬间隔效果在后续方法中可以用来展示。

当采用软间隔：

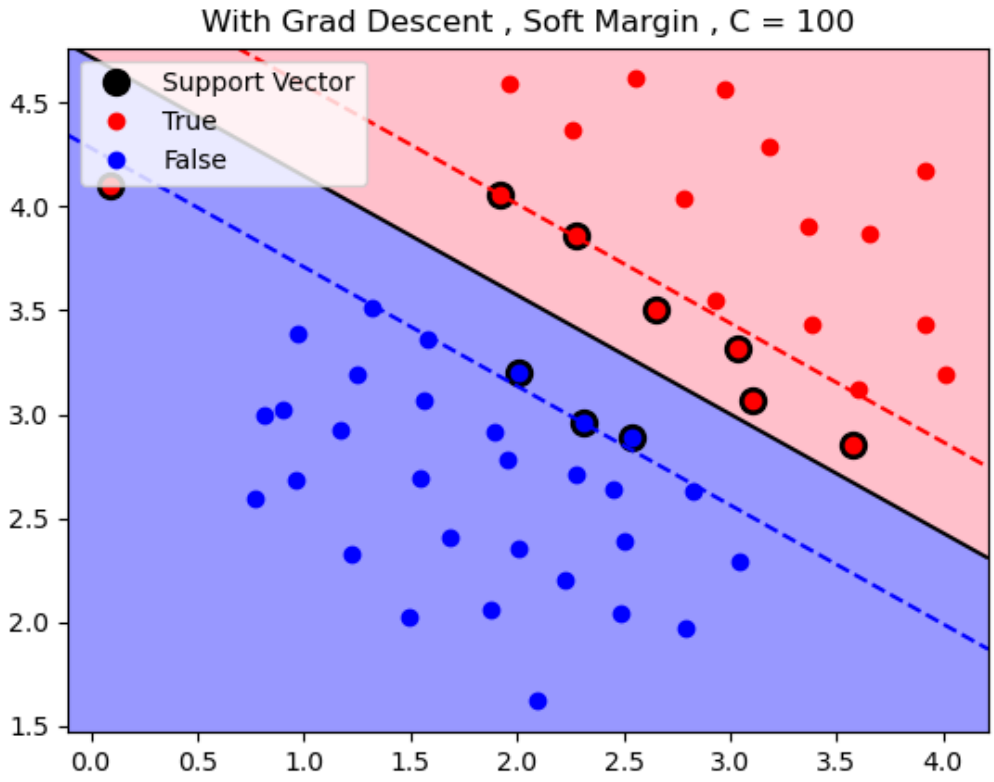
C=1时：



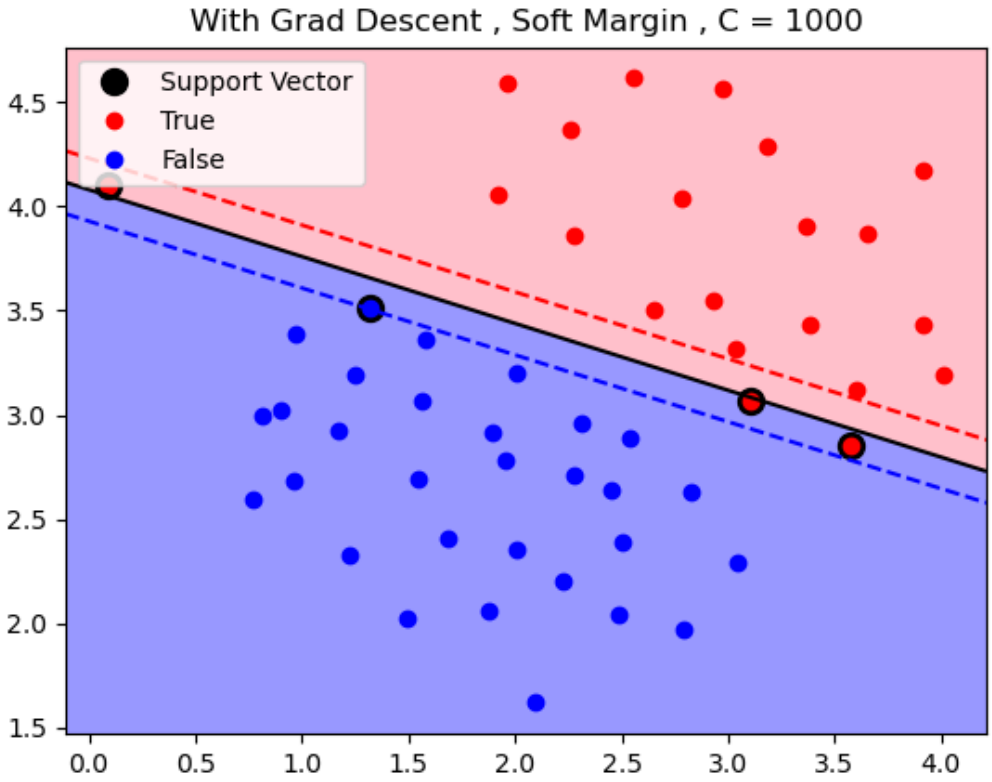
C=10时：


With Grad Descent , Soft Margin , C = 10
Support Vector
True
False
4.5
4.0
3.5
3.0
2.5
2.0
1.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

C=100时:


With Grad Descent , Soft Margin , C = 100
Support Vector
True
False
4.5
4.0
3.5
3.0
2.5
2.0
1.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

C=100时:



可以发现随着 惩罚因子C的增加，支持向量越来越少，而分类边界的间距越来越小。

2.7 对核函数下梯度下降的思考

在此基础上考虑是否能将该梯度下降方法来进行核函数操作。由于将一个函数进行核函数操作是一个升维度的过程，核函数的本事是将两个向量的点乘进行转换，但是绝大多数时候核函数对应单个函数的原函数并不能很好的计算，这样就导致升维后的函数：

$$y = \mathbf{w}^T \phi(\mathbf{x}) + b$$

由于升维后的$\phi(x)$ 的维度不能确定，那么对应的权重$\mathbf{w}$ 的维度也不好确定，那么这种情况下用如上的方法来进行梯度下降的过程并不能很好的使用。

但是是否可以考虑转换角度利用梯度下降呢？我个人首先认为这种方法的解析并不好，我先阐述我和我舍友的推断，我再阐述我个人对这种方法的想法。

首先先引入拉格朗日函数

$$L(\mathbf{w}, b; \lambda) = \frac{1}{2}||\mathbf{w}||^2 + \sum_{i=1}^{m} \lambda_i (1 - y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b))$$

$$\nabla_{\mathbf{w},b} L(\mathbf{w}, b; \lambda) = \mathbf{0}$$

$$\nabla_{\mathbf{w}} L(\mathbf{w}, b; \lambda) = \mathbf{w} - \sum_{i=1}^{m} \lambda_i y_i \phi(\mathbf{x}_i) = \mathbf{0} \quad \Rightarrow \quad \mathbf{w}^* = \sum_{i=1}^{m} \lambda_i y_i \phi(\mathbf{x}_i)$$

$$\nabla_b L(\mathbf{w}, b; \lambda) = \sum_{i=1}^{m} \lambda_i y_i = 0$$

在引入拉格朗日函数后的第一步为转换对应的约束问题，并把$\mathbf{w}^*$ 带回，在该函数的基础上再进行优化，即

$$minL(\mathbf{w},b;\lambda)=\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}y_iy_j\lambda_i\lambda_jK(\mathbf{x}_i,\mathbf{x}_j)+\sum_{i=1}^{m}\lambda_i(1-y_i(\sum_{j=1}^{m}\lambda_jy_jK(\mathbf{x}_i,\mathbf{x}_j)+b))$$

$$s.t.\sum_{i=1}^{m}\lambda_iy_i=0$$

然后再基于这个函数和合页损失进行求解

我个人认为在这个基础上进行对Λ和b进行梯度下降，一是回代后的函数我认为不并不可以直接采用，二是这个约束函数是强约束条件，那么如果我要在梯度下降中保持这个约束，那么在大多数情况下Λ的大多数都为0，三是拉格朗日处理后一般采用其对偶形式，不采用其对偶的形式总感觉存在问题，可能和新的约束问题和原问题并不包容。

3. 基于拉格朗日对偶的暴力SVM（二次规划）与核函数

3.1 基础二次规划解决 Λ 取值

正如上文所描述，原问题的拉格朗日对偶形式如下：

$$\begin{aligned}\max_{\mathbf{\Lambda}}\quad & \mathbf{1}^T\mathbf{\Lambda}-\frac{1}{2}\mathbf{\Lambda}^TD\mathbf{\Lambda}\\ s.t.\quad & y^T\mathbf{\Lambda}=\mathbf{0}\\ & \mathbf{\Lambda}\geq 0\end{aligned}$$

在引入惩罚因子C后的拉格朗日对偶函数形式如下：

$$\begin{aligned}\max_{\mathbf{\Lambda}}\quad & \mathbf{1}^T\mathbf{\Lambda}-\frac{1}{2}\mathbf{\Lambda}^TD\mathbf{\Lambda}\\ s.t.\quad & y^T\mathbf{\Lambda}=\mathbf{0}\\ & 0\leq\mathbf{\Lambda}\leq C\end{aligned}$$

显然这个一个二次规划问题，有着许多的解法，由于我需要最适用的方法来求解这个二次规划问题，并且用这个最普通的方法来验证我另外两种做法是否正确，因此我引入了python中的`cvxopt`包。

`cvxopt`是一个用于解决优化问题的Python包，特别擅长处理凸优化问题。凸优化问题是指目标函数是凸函数且约束条件构成凸集的问题。这个包提供了多种优化算法和工具，广泛应用于线性规划（LP）、二次规划（QP）、半定规划（SDP）、以及其他类型的优化问题。

调用`cvxopt.solvers.qp()`方法即可求解该二次规划问题。调用`cvxopt.solvers.qp()`方法需要将原对偶问题转换为如下的标准形式：

$$\begin{aligned}min\quad & \frac{1}{2}\mathbf{x}^TP\mathbf{x}+\mathbf{q}^T\mathbf{x}\\ s.t.\quad & G\mathbf{x}\leq\mathbf{h}\\ & A\mathbf{x}=\mathbf{b}\end{aligned}$$

若引入惩罚因子，则在将其转换为两个不等式约束，在进行求解后，$\lambda_i\neq 0$ 时，对应的向量为支持向量，因此要将支持向量保存，用来后续求解：

$$y=\sum_{i=1}^{sv}\lambda_iy_i\mathbf{x}_i^T\mathbf{x}+b$$

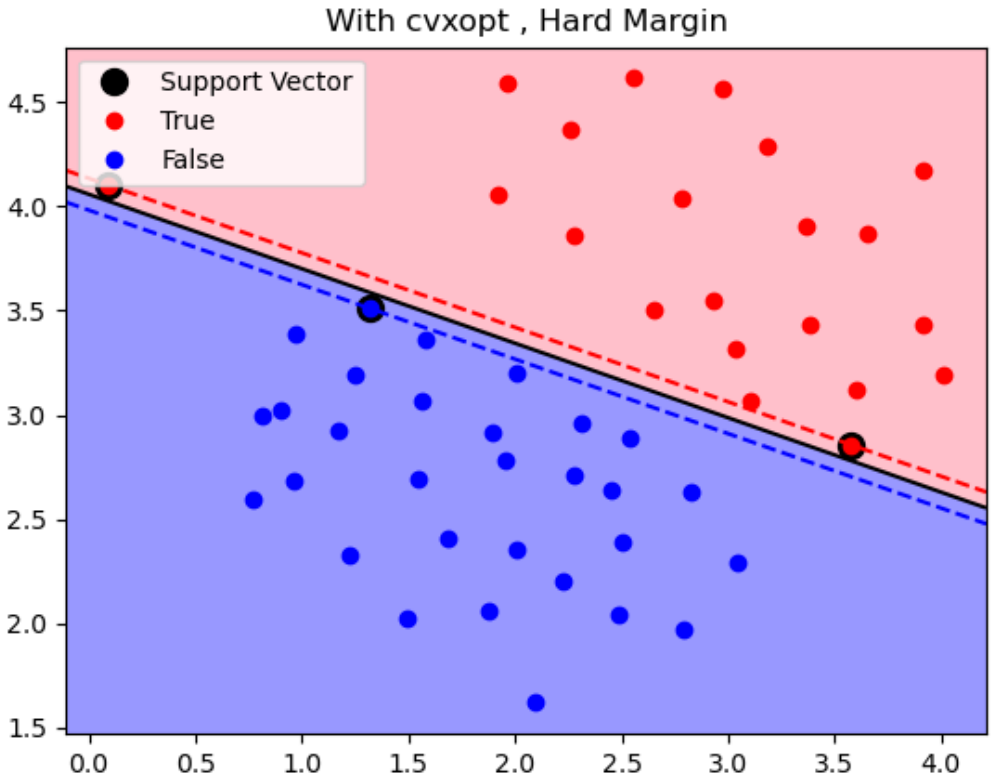
上述式子中均为支持向量，因此保存所有的支持向量而不是保存系数（为了后续升维统一）

```
    def fitByStd(self, X, label, margin='soft', kernel='none', gamma=10):
        self.stategy = 'std'
        self.kernel = kernel
```

```
        self.gamma = gamma
        self.trained = True
        X = X.values
        n_samples, n_features = X.shape
        # print(n_samples, n_features)
        y = label.astype(float)
        y[label == 0] = -1
        y = np.array(y).reshape(-1, 1)

        K = np.dot(X, X.T)
        P = np.dot(y, y.T) * K
        q = np.ones(n_samples) * -1
        A = y.reshape(1, -1)
        b = np.zeros(1)
        if margin == 'hard':
            G = -np.eye(n_samples)
            h = np.zeros(n_samples)
        if margin == 'soft':
            G = np.vstack((-np.eye(n_samples), np.eye(n_samples)))
            h = np.hstack((np.zeros(n_samples), np.ones(n_samples) *
self.C))

        P = cvxopt.matrix(P)
        q = cvxopt.matrix(q)
        G = cvxopt.matrix(G)
        h = cvxopt.matrix(h)
        A = cvxopt.matrix(A)
        b = cvxopt.matrix(b)

        solution = cvxopt.solvers.qp(P, q, G, h, A, b)
        self.Lambda = np.ravel(solution['x'])
        # print(self.Lambda)
        sv = self.Lambda > 1e-5
        index = np.arange(len(self.Lambda))[sv]
        self.Lambda = self.Lambda[sv]
        sv_X = X[sv]
        sv_y = y[sv]
        self.sv_X = sv_X
        self.sv_y = sv_y

        temp_y = np.zeros(self.sv_X.shape[0]).reshape(-1, 1)
        for i in range(len(self.Lambda)):
            if self.kernel == 'none':
                temp_y += self.Lambda[i] * sv_y[i] * np.dot(sv_X,
sv_X[i].T).reshape(-1, 1)
            if self.kernel == 'rbf':
                temp_y += self.Lambda[i] * sv_y[i] * rbf_kernel(sv_X,
sv_X[i].T, gamma).reshape(-1, 1)
        self.bias = np.mean(sv_y - temp_y)
        return self.sv_X
```

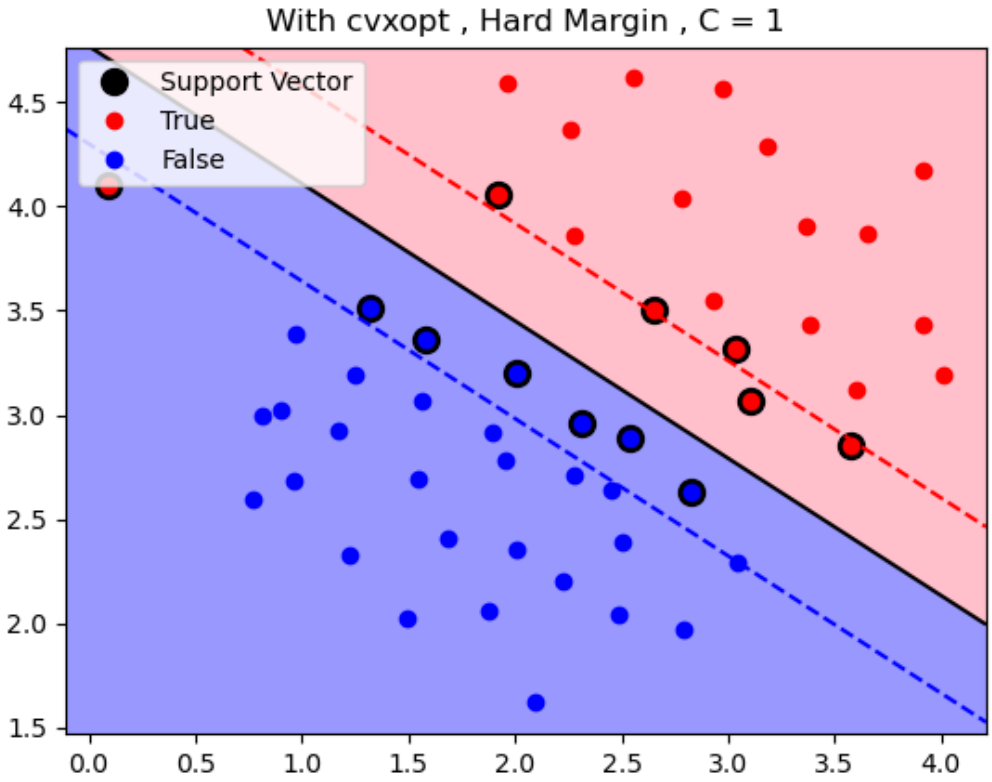
随后对该方法下进行预测和可视化展示：

采取硬间隔时：

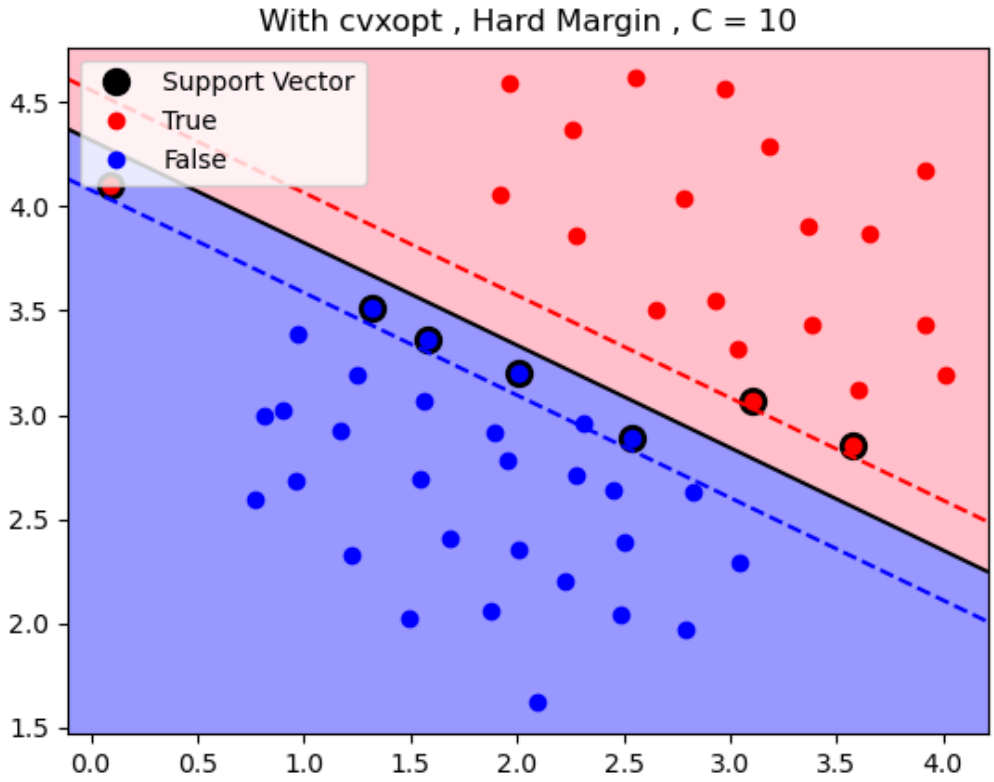


采取软间隔

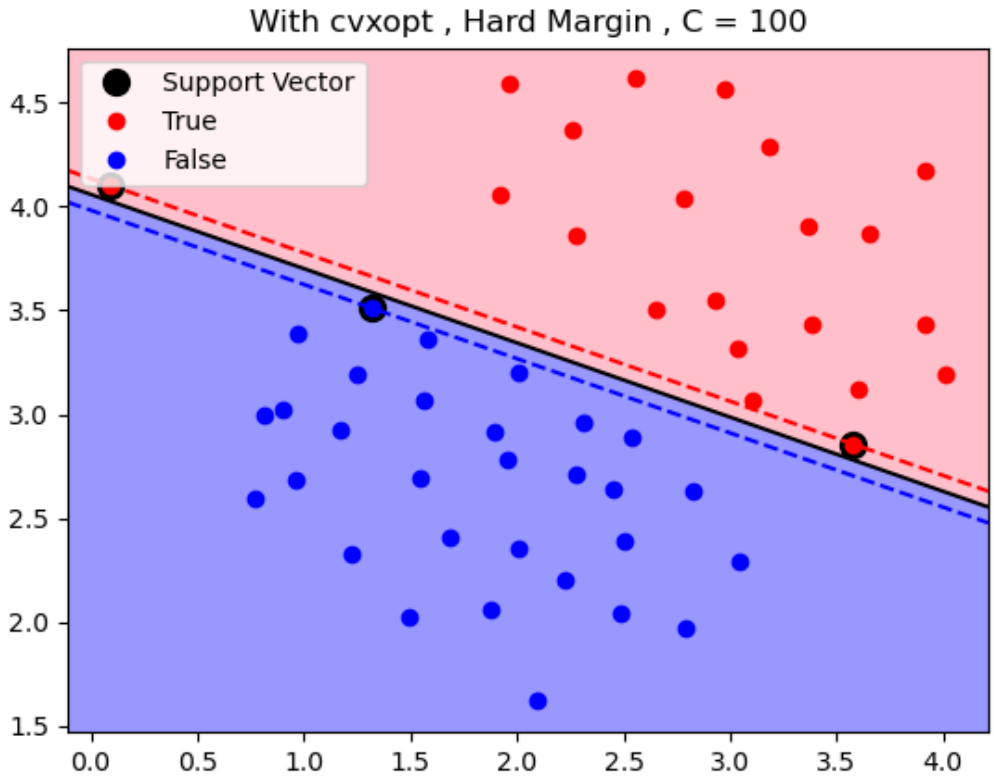
C=1时：



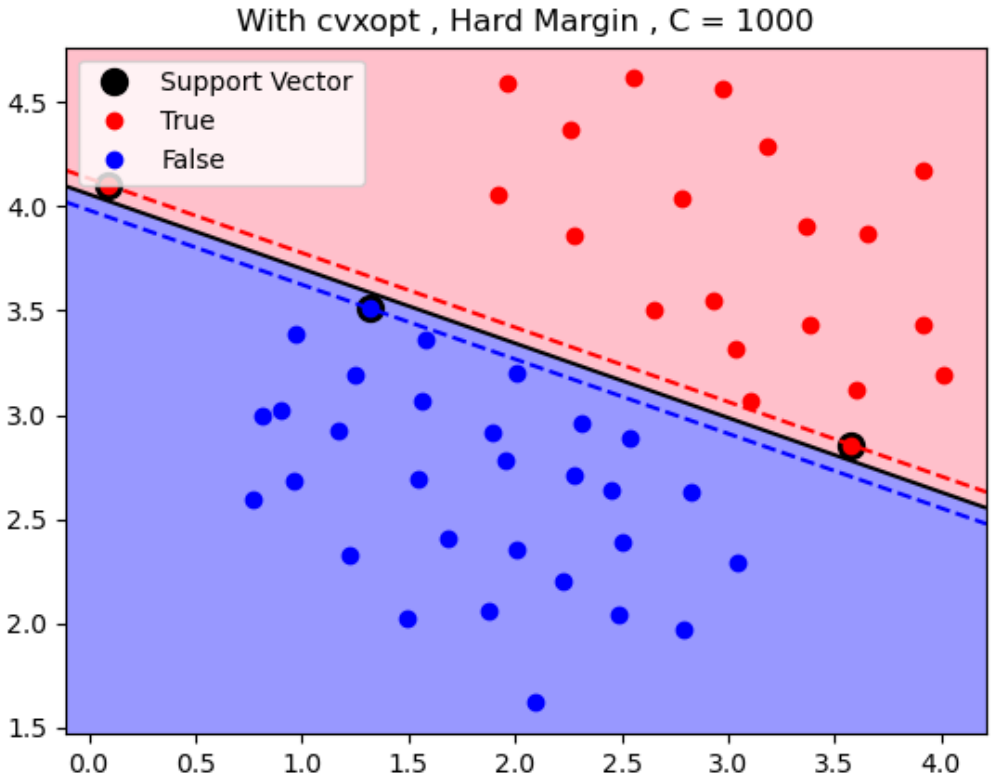
C=10时：



C=100时：



C=1000时：



可见该方法的对C的收束速度很快，在C=100时已经和硬间隔没有区别。

3.2 引入核函数

这个方法引入核函数相当的简单，因为该之前写好的方法中并不是更新权重，而是保存的支持向量，并且对应的所有向量点乘的工作都是用一个线性核构成的，因此需要修改的点仅在定义对应核函数的实现方式和根据初始化进行调用。

在对应的 `fit` 函数中增加修改为：

```
        if kernel == 'none':
            K = np.dot(X, X.T)
        if kernel == 'rbf':
            K = rbf_kernel(X, X, gamma)
```

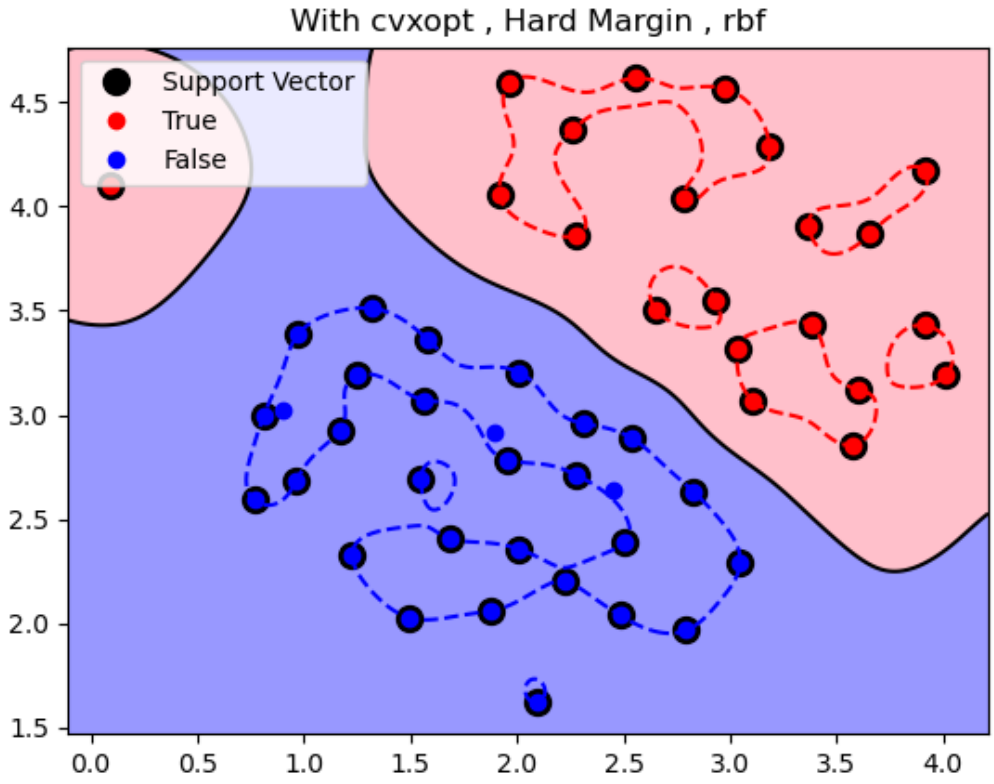
并且定义高斯核函数：

```
def rbf_kernel(X1, X2, gamma=0.1):
    if X1.ndim == 1 and X2.ndim == 1:
        return np.exp(-gamma * np.linalg.norm(X1 - X2) ** 2)
    elif X1.ndim > 1 and X2.ndim == 1:
        return np.exp(-gamma * np.linalg.norm(X1 - X2, axis=1) ** 2)
    elif X1.ndim == 1 and X2.ndim > 1:
        return np.exp(-gamma * np.linalg.norm(X1 - X2, axis=1) ** 2)
    elif X1.ndim > 1 and X2.ndim > 1:
        return np.exp(-gamma * np.linalg.norm(X1[:, np.newaxis] -
X2[np.newaxis, :], axis=2) ** 2)
```

这样引入后即可在该方法上实现核函数进行升维度。

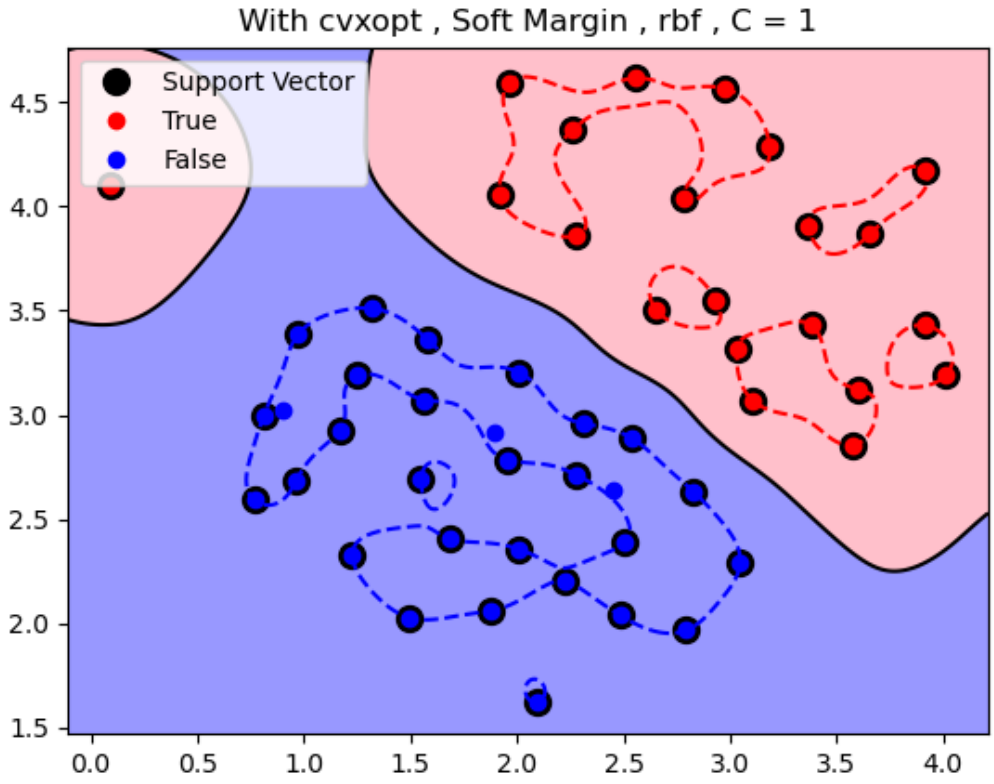
3.3 对第一个数据进行核函数应用

采用硬间隔：过拟合十分严重

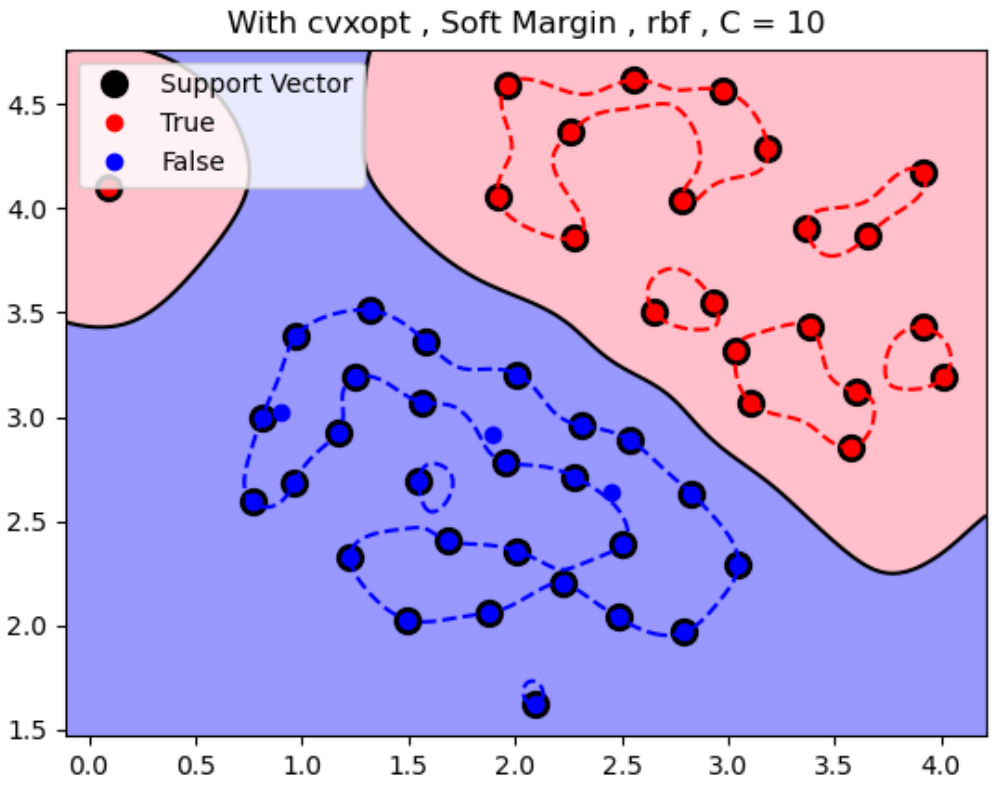


采用软间隔

C = 1 :



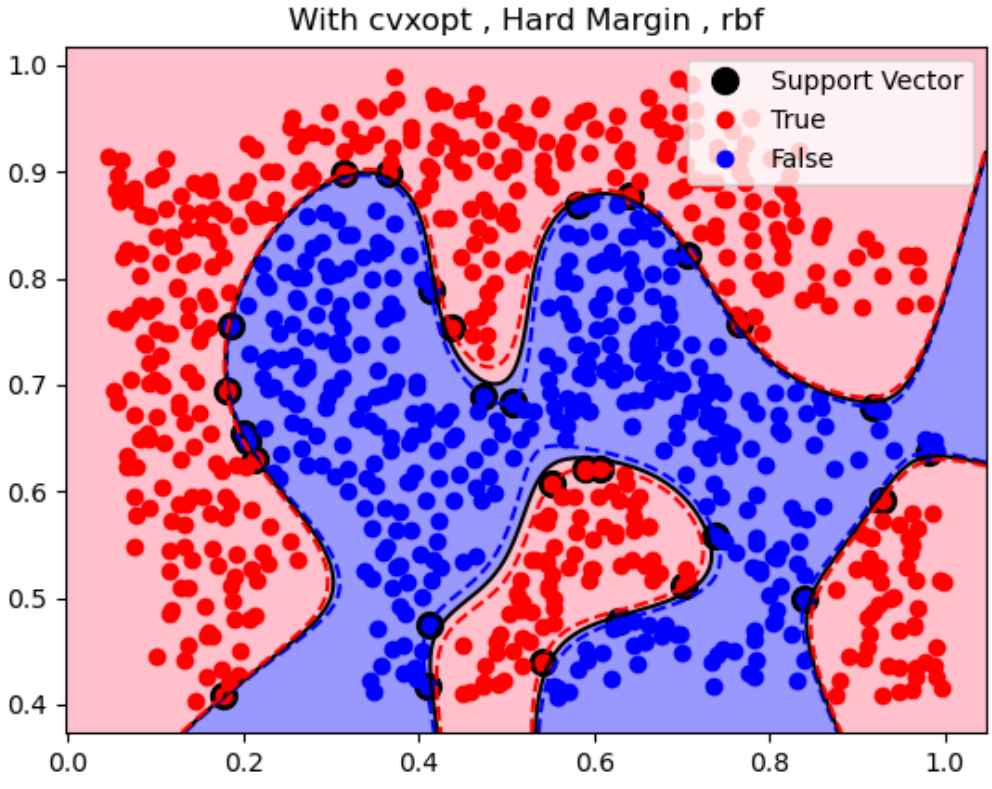
C = 10：



可以看出来，由于原内容线性可分，在升维度后有无间隔都可以很好的保持分隔。

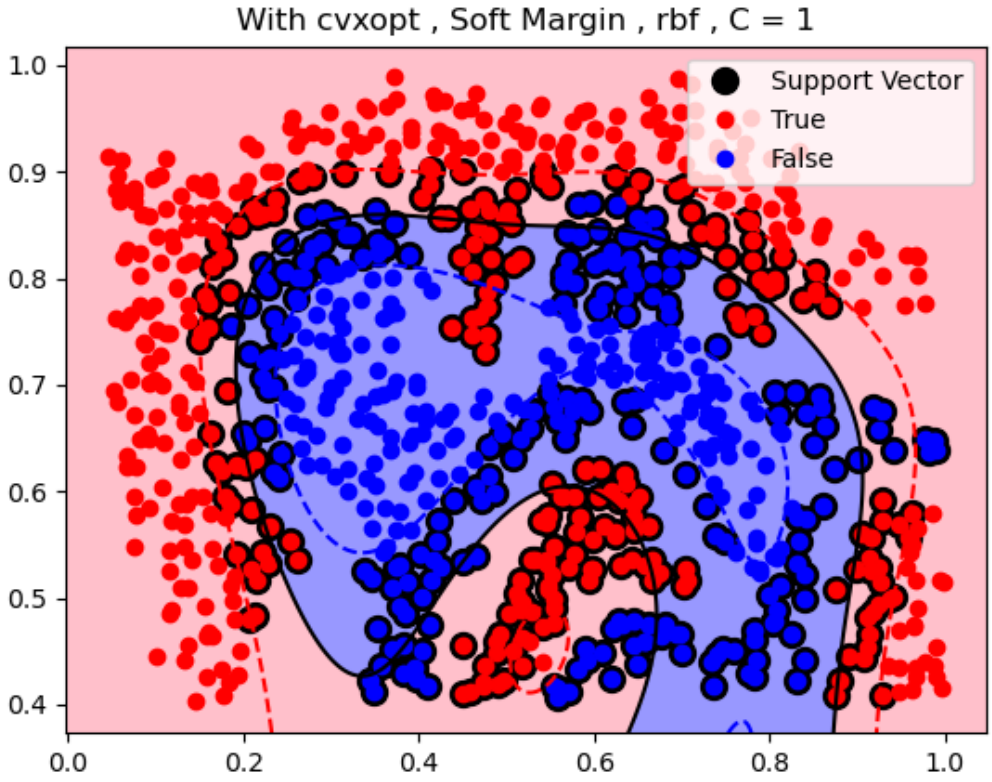
3.4 对第二个数据进行核函数应用

采用硬间隔：

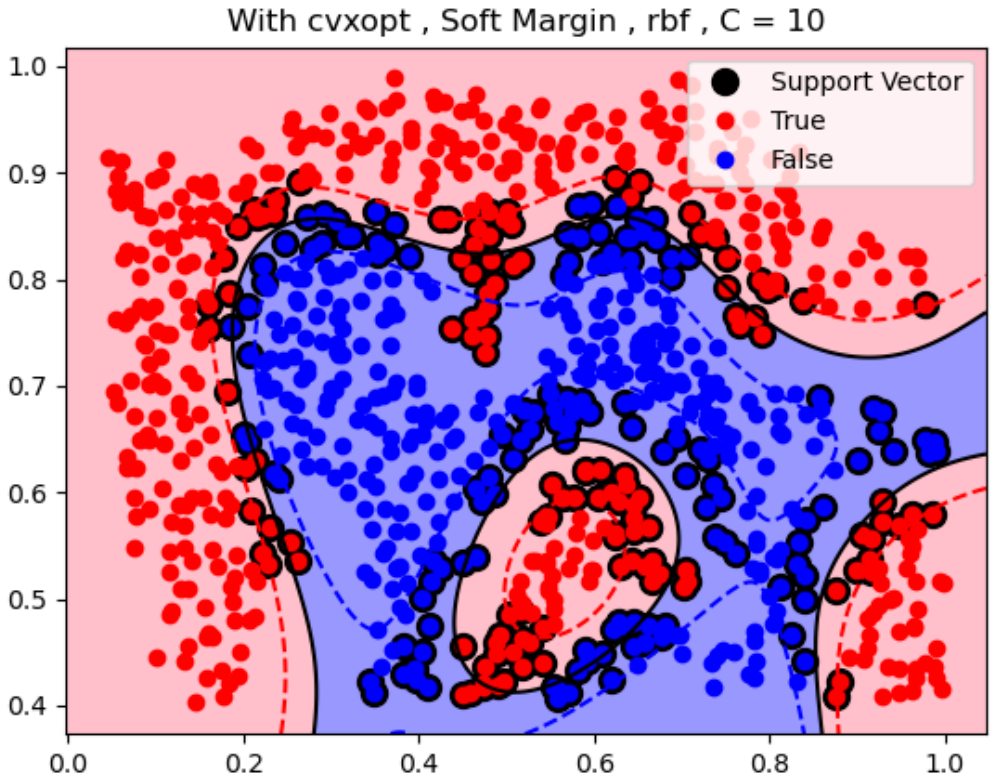


采用软间隔：

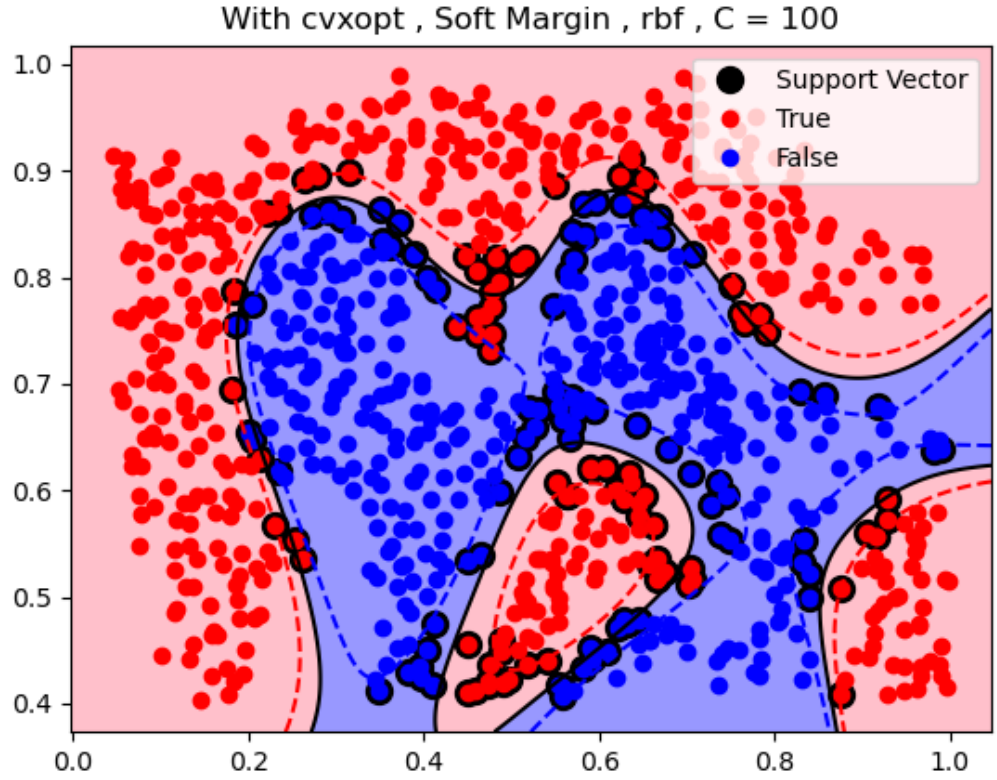
C = 1：



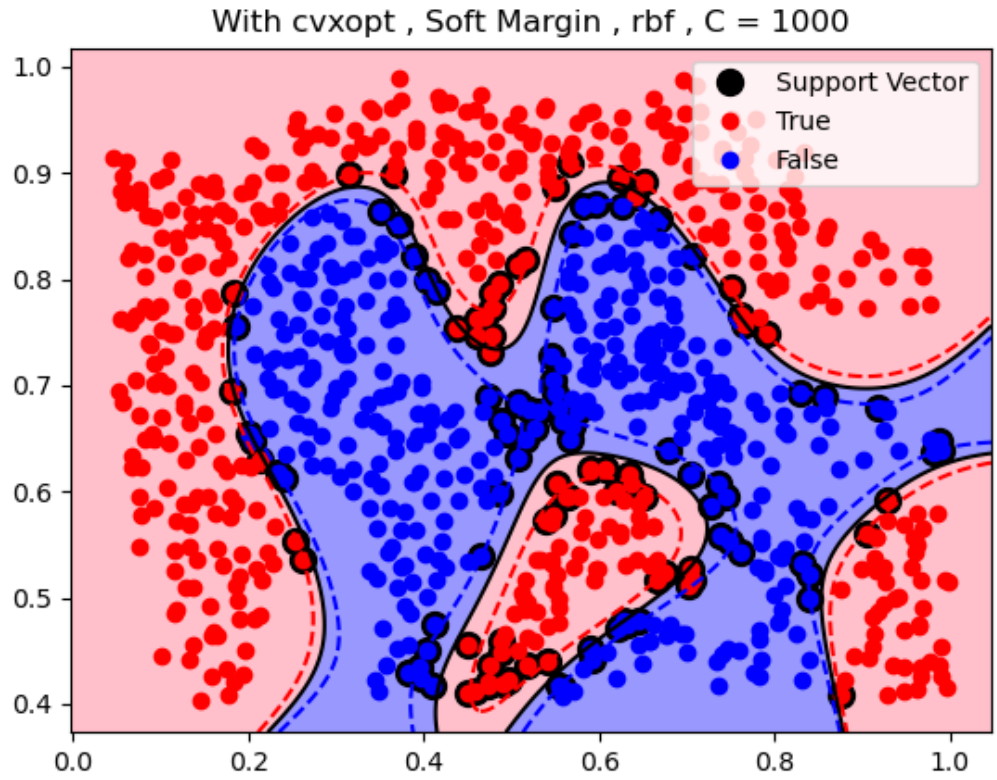
C = 10：



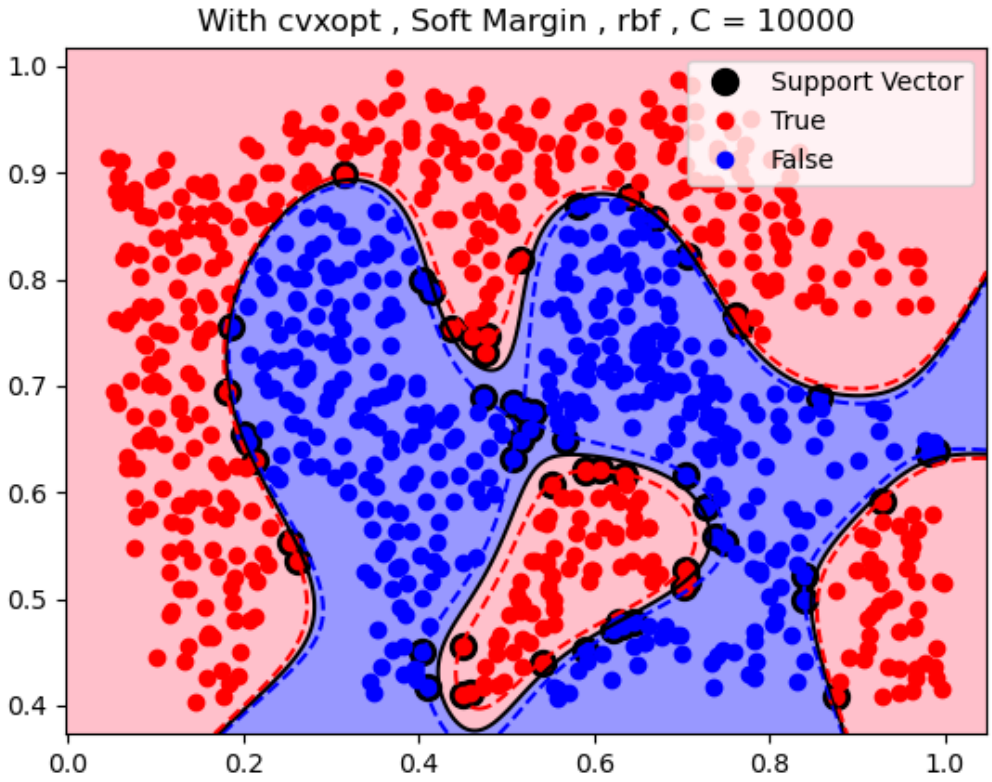
C = 100：



C = 1000:



C = 10000:



4. 基于SMO的SVM与核函数实现

虽然上述才哟个`cvxopt`的效果很好，但是本质其还是调用了其他包来实现，为了更好更快的求解二次规划，在此引入SMO算法。

4.1 SMO理论

正如上述软间隔所描述的约束优化表达式

$$\max_{\Lambda} \quad \sum_{i=1}^{m}\lambda_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\lambda_i\lambda_j y_i y_j K(\mathbf{x}_i,\mathbf{x}_j))$$

$$s.t. \quad \sum_{i=1}^{m}\lambda_i y_i = 0$$

$$0 \le \lambda_i \le C, \quad i = 1,2,\dots,m$$

对KKT 条件进行一个简单的转换，则会得到：

$$\lambda_i = 0 \Leftrightarrow y_i f(\mathbf{x}_i) \ge 1$$
$$0 < \lambda_i < C \Leftrightarrow y_i f(\mathbf{x}_i) = 1$$
$$\lambda_i = C \Leftrightarrow y_i f(x_i) \le 1$$

在 SMO（序列最小化）方法出现之前，人们依赖于二次规划求解工具（例如基于内点法的`cvxopt`）来解决上述的优化问题，训练 SVM。这些工具需要具有强大计算能力的计算机进行支撑，实现也比较复杂。SMO 算法将优化问题分解为容易求解的若干小的优化问题，来训练 SVM。简言之，SMO 仅关注 Λ**对** 和 **偏置 b 的求解更新，进而求解出权值向量 $\mathbf{w}$的隐式表达，得到决策边界（分割超平面），从而大大减少了运算复杂度。(理论上会减少运算量和复杂度)

SMO会选择一对λ_i 和 λ_j，并固定其他参数（将其他参数认为是常数），则约束条件就会变为：

$$\lambda_i y_i + \lambda_j y_j = -\sum_{k \neq i,j} \lambda_k y_k$$
$$0 \leq \lambda_i \leq C$$
$$0 \leq \lambda_j \leq C$$

那么原问题可以替换为：

$$\max_{\lambda_i, \lambda_j} \quad (\lambda_i + \lambda_j) - [\frac{1}{2} K_{ii} \lambda_i^2 + \frac{1}{2} K_{jj} \lambda_j^2 + y_i y_j K_{ij} \lambda_i \lambda_j] - [y_i \lambda_i \sum_{k \neq i,j} y_k \lambda_k K_{ki} + y_j \lambda_j \sum_{k \neq i,j} y_k \lambda_k K_{kj}]$$
$$s.t. \quad \lambda_i y_i + \lambda_j y_j = -\sum_{k \neq i,j} \lambda_k y_k$$
$$0 \leq \lambda_i \leq C$$
$$0 \leq \lambda_j \leq C$$

从数学上来看，λ_i, λ_j 的取值 要落在 0-C取值的正方形中，且要落在对应的一条线段上。

若放缩λ_i 的取值边界，则有 $L \leq \lambda_i \leq H$

由此可以求出 λ_i 的上下界：

- $y_i \neq y_j$ 时：

$$L = max(0, \lambda_i - \lambda_j), \quad H = min(C, C + \lambda_j - \lambda_i)$$

- $y_i = y_j$ 时：

$$L = max(0, C + \lambda_j - \lambda_i), \quad H = min(C, \lambda_j + \lambda_i)$$

定义优化函数为：

$$\Psi = (\lambda_i + \lambda_j) - [\frac{1}{2} K_{ii} \lambda_i^2 + \frac{1}{2} K_{jj} \lambda_j^2 + y_i y_j K_{ij} \lambda_i \lambda_j]$$

由于λ_i 和 λ_j 有线性关系，因此可以消去λ_i，进而令 Ψ 对 λ_j 求二阶导，并令二阶导为0可以得到：

$$\lambda_{jnew}(2K_{ij} - K_{ii} - K_{jj}) = \lambda_{jold}(2K_{ij} - K_{ii} - K_{jj}) - y_j(E_i - E_j)$$
$$E_i = f(\mathbf{x}_i) - y_i$$

令：$\eta = 2K_{ij} - K_{ii} - K_{jj}$

因此：

$$\lambda_{jnew} = \lambda_{jold} - \frac{y_j(E_i - E_j)}{\eta}$$

但是还需要考虑上下界截断：

$$\lambda_{jnewclipped} = \begin{cases} H, & if \quad \lambda_{jnew} \geq H \\ \lambda_{jnew}, & if \quad L < \lambda_{jnew} < H \\ L, & if \quad \lambda_{jnew} \leq L \end{cases}$$

从而得到 λ_i 的更新：

$$\lambda_{inew} = \lambda_{iold} + y_i y_j (\lambda_{jold} - \lambda_{jnewclipped})$$

令：

$$b_1 = b_{old} - E_i - y_i K_{ii}(\lambda_{inew} - \lambda_{iold}) - y_j K_{ij}(\lambda_{jnewclipped} - \lambda_{jold})$$
$$b_2 = b_{old} - E_j - y_i K_{ij}(\lambda_{inew} - \lambda_{old}) - y_j K_{jj}(\lambda_{jnewclipped} - \lambda_{jold})$$

则 b 的更新为：

$$b_{new} = \begin{cases} b_1, & if \quad 0 < \lambda_{inew} < C \\ b_2, & if \quad 0 < \lambda_{jnewclipped} < C \\ \frac{b_1+b_2}{2}, & otherwise \end{cases}$$

4.2 启发式选择

如果两个拉格朗日乘子其中之一违背了 KKT 条件，此时，每一次乘子对的选择，都能使得优化目标函数减小。

若 $\lambda_i = 0$，可知样本 $\mathbf{x}_i$ 不会对模型产生影响

若 $\lambda_i = C$，样本 $\mathbf{x}_i$ 不会是支持向量

若 $0 < \lambda_i < C$，则λ_i 没有落在边界上，当下式满足时，λ_i 会违反KKT条件：

$$\lambda_i < C \quad and \quad y_i f(\mathbf{x}_i) - 1 < 0$$
$$\lambda_i > 0 \quad and \quad y_i f(\mathbf{x}_i) - 1 > 0$$

由于上式过于严苛，可以考虑设置一个容忍区间 $[-\tau, \tau]$，并考虑令：

$$R_i = y_i E_i = y_i(f(\mathbf{x}_i) - y_i) = y_i f(\mathbf{x}_i) - 1$$

可以将违反KKT条件表达式写为：

$$\lambda_i < C \quad and \quad R_i < -\tau$$
$$\lambda_i > 0 \quad and \quad R_i > \tau$$

则启发式选择λ_i, λ_j 可以看作两层循环：

外层循环中，如果当前没有 λ 对的变化，意味着所有的 λ_i 都遵循了KKT条件，需要在整个样本集上进行迭代。否则，只需要选择在处在边界内 $0 < \lambda_i < C$、并且违反了KKT条件的λ_i。

内层循环中，选出使得 $E_i - E_j$ 达到最大的 λ_j

4.3 代码实现

已经引入了核函数的实现。

```
    def fitBySMO(self, X, label, C=1.0, margin='soft', kernel='none',
gamma=10, tolerance=1e-6, max_iter=1000):
        self.trained = True
        self.kernel = kernel
        self.gamma = gamma
        self.stategy = 'smo'
        X = X.values
        n_samples, n_features = X.shape
        y = label.astype(float)
        y[label == 0] = -1
        y = np.array(y).reshape(-1, 1)
        self.Lambda = np.zeros((n_samples, 1))
        self.bias = 0

        # 缓存核计算结果
        if self.kernel == 'none':
```

```
            K = np.dot(X, X.T)
        elif self.kernel == 'rbf':
            K = rbf_kernel(X, X, gamma=self.gamma)
        else:
            K = np.dot(X, X.T)  # 默认使用线性核

        def compute_error(i):
            fx_i = np.dot((self.Lambda * y).T, K[:, i]) + self.bias
            E_i = fx_i - y[i]
            return E_i

        def choose_alpha_pair(i, E_i):
            non_bound_indices = np.where((self.Lambda > 0) & (self.Lambda < C))[0]
            if len(non_bound_indices) > 1:
                if E_i > 0:
                    j = np.argmin([compute_error(k) for k in non_bound_indices])
                else:
                    j = np.argmax([compute_error(k) for k in non_bound_indices])
                j = non_bound_indices[j]
                return j
            else:
                j = np.random.choice(list(set(range(n_samples)) - {i}))
                return j

        def update_alpha_pair(i, j):
            E_i = compute_error(i)
            E_j = compute_error(j)
            alpha_i_old = self.Lambda[i].copy()
            alpha_j_old = self.Lambda[j].copy()

            if y[i] != y[j]:
                L = max(0, alpha_j_old - alpha_i_old)
                H = min(C, C + alpha_j_old - alpha_i_old)
            else:
                L = max(0, alpha_i_old + alpha_j_old - C)
                H = min(C, alpha_i_old + alpha_j_old)

            if L == H:
                return False

            eta = 2 * K[i, j] - K[i, i] - K[j, j]
            if eta >= 0:
                return False

            self.Lambda[j] -= y[j] * (E_i - E_j) / eta
            self.Lambda[j] = max(L, min(H, self.Lambda[j]))

            self.Lambda[i] += y[i] * y[j] * (alpha_j_old - self.Lambda[j])

            b1 = self.bias - E_i - y[i] * (self.Lambda[i] - alpha_i_old) * K[i, i] - y[j] * (
                    self.Lambda[j] - alpha_j_old) * K[i, j]
            b2 = self.bias - E_j - y[i] * (self.Lambda[i] - alpha_i_old) * K[i, j] - y[j] * (
                    self.Lambda[j] - alpha_j_old) * K[j, j]
```

```
            if 0 < self.Lambda[i] < C:
                self.bias = b1
            elif 0 < self.Lambda[j] < C:
                self.bias = b2
            else:
                self.bias = (b1 + b2) / 2
            return True

        iteration = 0
        while iteration < max_iter:
            num_changed = 0
            for i in range(n_samples):
                E_i = compute_error(i)
                if (y[i] * E_i < -tolerance and self.Lambda[i] < C) or (y[i] * E_i > tolerance and self.Lambda[i] > 0):
                    j = choose_alpha_pair(i, E_i)
                    if update_alpha_pair(i, j):
                        num_changed += 1
            if num_changed == 0:
                break
            iteration += 1

        sv_indices = np.where(self.Lambda > 1e-5)[0]
        self.sv_X = X[sv_indices]
        self.sv_y = y[sv_indices]
        self.Lambda = self.Lambda[sv_indices]

        return self.sv_X
```

4.4 代码特殊处理点

以下两端代码段实现了SMO算法的两个不同版本，时间差异巨大的原因在于它们采用了不同的启发式方法来选择需要优化的拉格朗日乘子对，并且处理方式有所不同。

第一段循环：

```
while num_changed > 0 or examine_all:
    num_changed = 0
    if examine_all:
        for i in range(n_samples):
            _, j, E_i, E_j = choose_alpha_pair(i)
            if j is not None:
                if update_alpha_pair(i, j, E_i, E_j):
                    num_changed += 1
    else:
        for i in range(n_samples):
            if 0 < self.Lambda[i] < C:
                _, j, E_i, E_j = choose_alpha_pair(i)
                if j is not None:
                    if update_alpha_pair(i, j, E_i, E_j):
                        num_changed += 1
    examine_all = (examine_all == False)
```

特点：

1. **检查所有样本（examine_all = True）**：在这种情况下，所有样本都会被检查，时间复杂度较高，因为每个样本都需要进行一次拉格朗日乘子的选择和更新。
2. **只检查非边界样本（examine_all = False）**：在这种情况下，只检查那些拉格朗日乘子在（0, C）范围内的样本，减少了需要检查的样本数。

时间差异原因：

- 当`examine_all`为True时，每次迭代都会遍历所有样本，消耗大量时间。
- 当`examine_all`为False时，只遍历非边界样本，减少了遍历次数，但如果条件不满足，需要反复切换`examine_all`状态，导致额外的迭代和检查。

第二段循环：

```
iteration = 0
while iteration < max_iter:
    num_changed = 0
    for i in range(n_samples):
        E_i = compute_error(i)
        if (y[i] * E_i < -tolerance and self.Lambda[i] < C) or (y[i] * E_i > tolerance and self.Lambda[i] > 0):
            j = choose_alpha_pair(i, E_i)
            if update_alpha_pair(i, j):
                num_changed += 1
    if num_changed == 0:
        break
    iteration += 1
```

特点：

1. **固定迭代次数（max_iter）**：每次迭代遍历所有样本，但通过`max_iter`限制最大迭代次数。
2. **直接选择对违反KKT条件的样本对进行优化**：减少不必要的迭代次数。

时间差异原因：

- 每次迭代都遍历所有样本，但直接对违反KKT条件的样本对进行优化，这样在每次迭代中更高效。
- 在更新拉格朗日乘子对时，如果没有发生变化（num_changed == 0），则直接退出循环，减少了不必要的迭代次数。

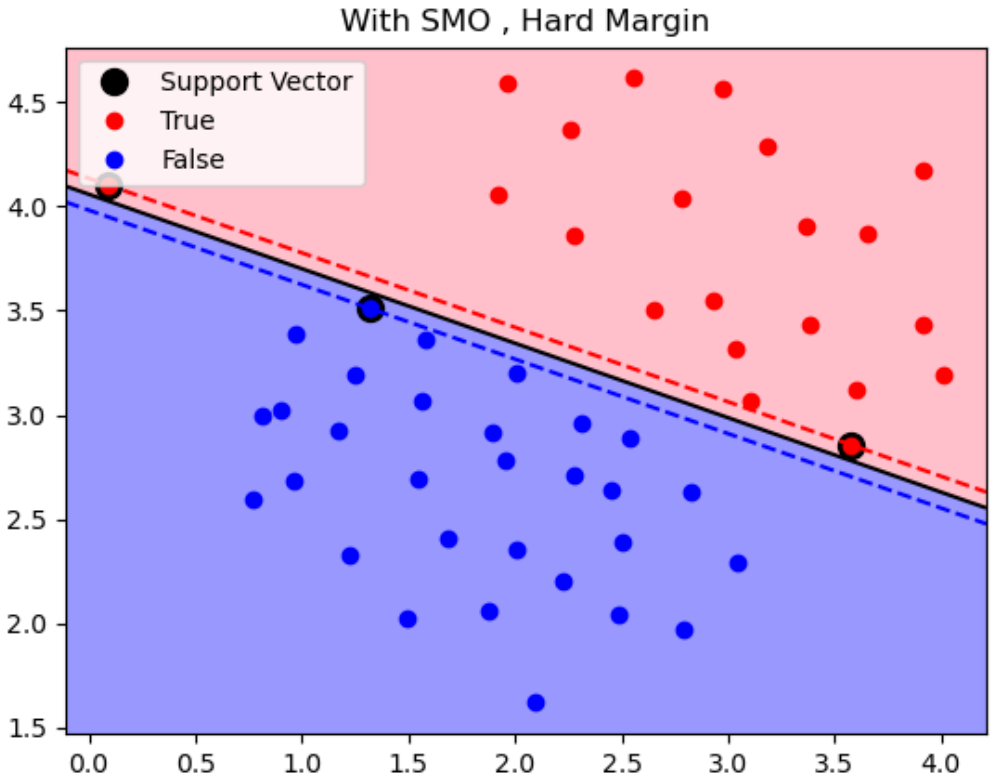
第一段循环：采用检查所有样本和非边界样本交替进行的方法，导致在每次迭代中都可能进行大量无效的检查，从而影响性能。

第二段循环：每次迭代都遍历所有样本，但只对违反KKT条件的样本对进行优化，同时通过设置最大迭代次数限制总的迭代次数，优化了算法的效率。

因此，第二段代码在时间复杂度和效率上更优，因为它减少了不必要的检查和迭代次数，更快地收敛到解决方案。

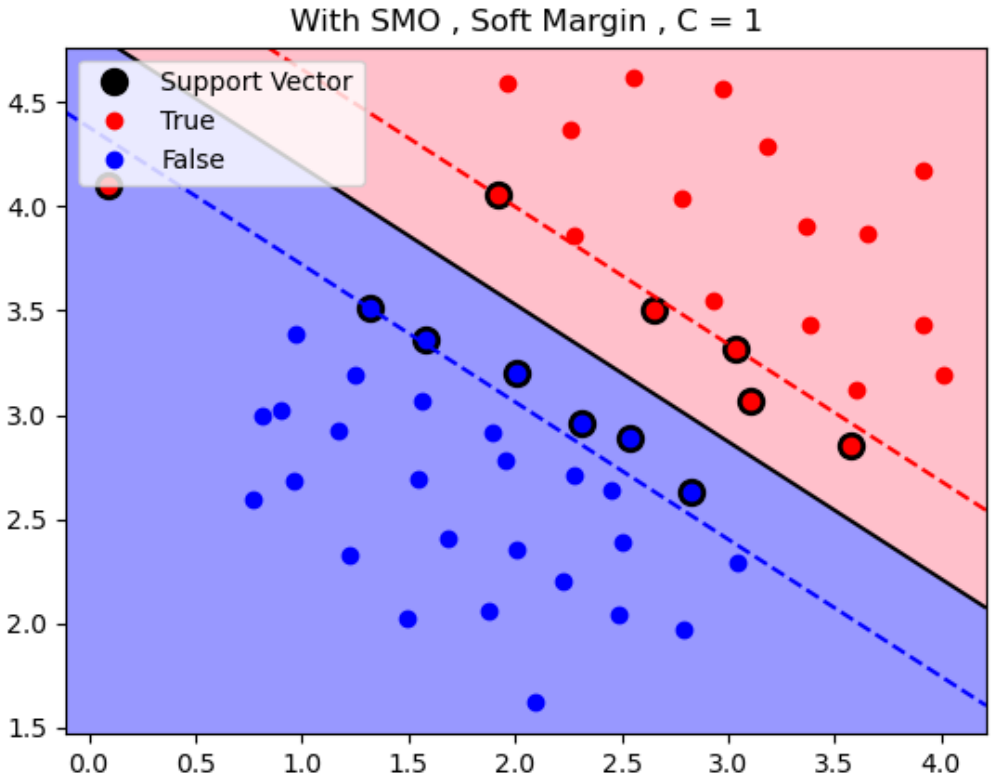
4.5 第一组数据无核函数实现

采用硬间隔：

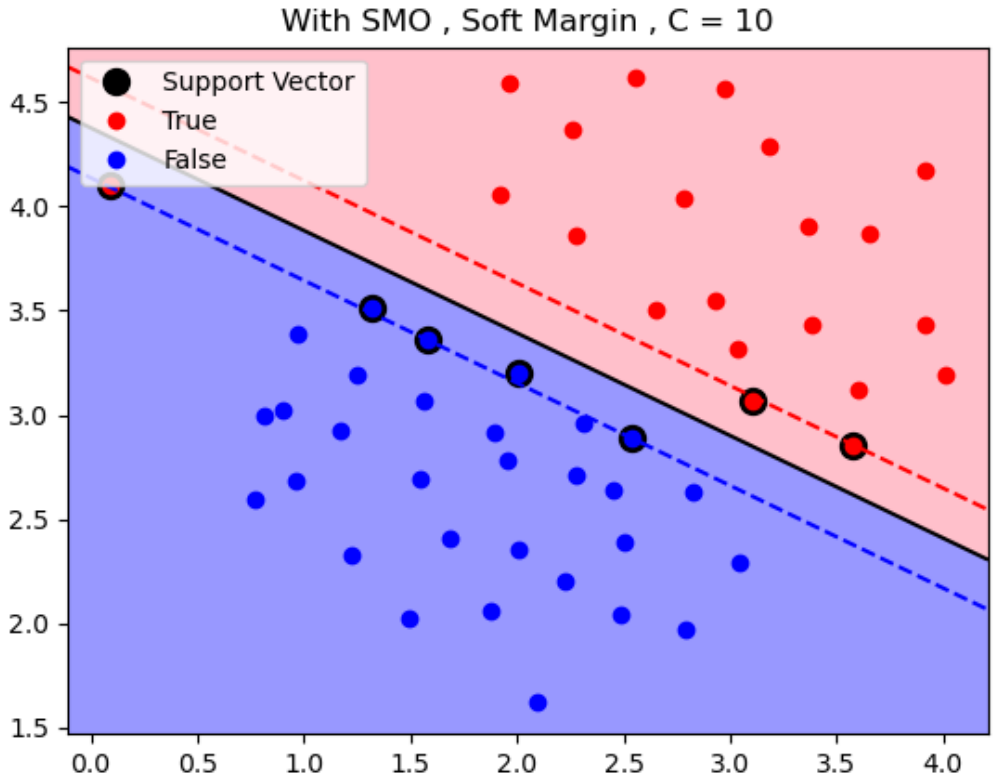


采用软间隔：可以发现C=100以上时与硬间隔保持一致

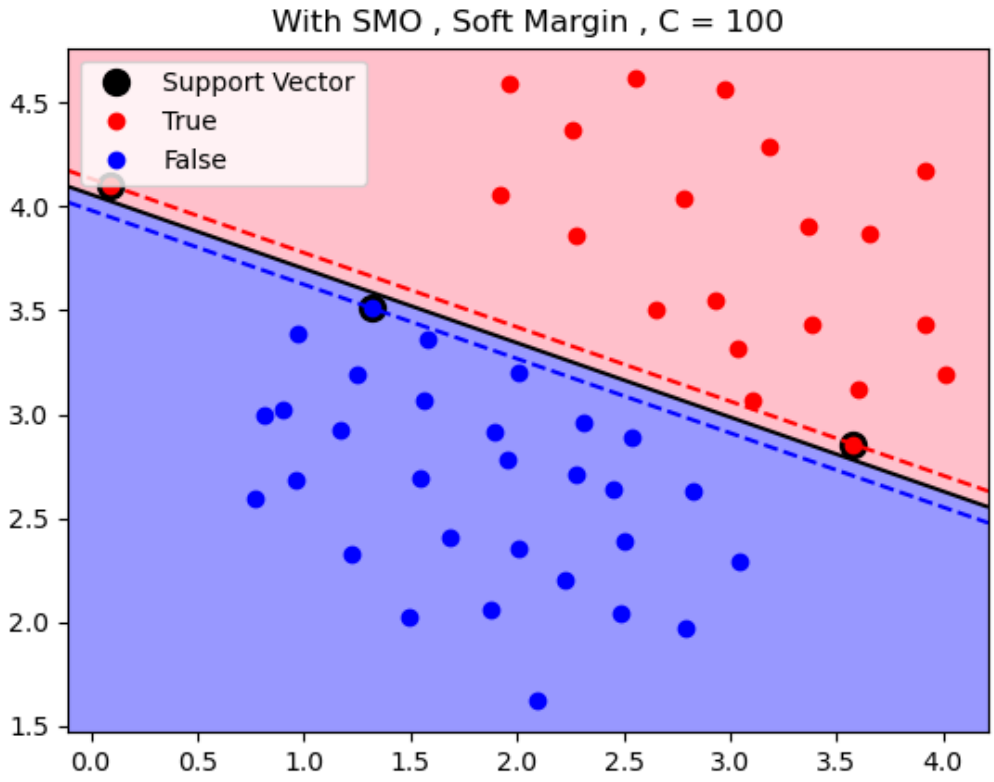
C = 1：



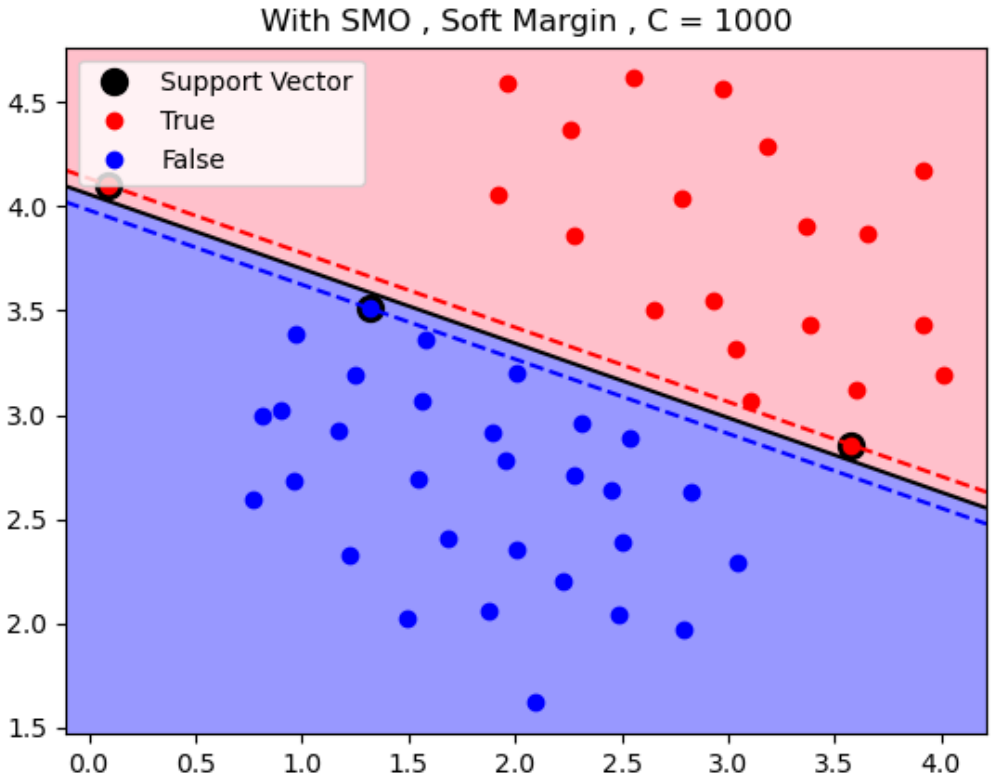
C = 10：


With SMO , Soft Margin , C = 10
Support Vector
True
False
4.5
4.0
3.5
3.0
2.5
2.0
1.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

C = 100:


With SMO , Soft Margin , C = 100
Support Vector
True
False
4.5
4.0
3.5
3.0
2.5
2.0
1.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

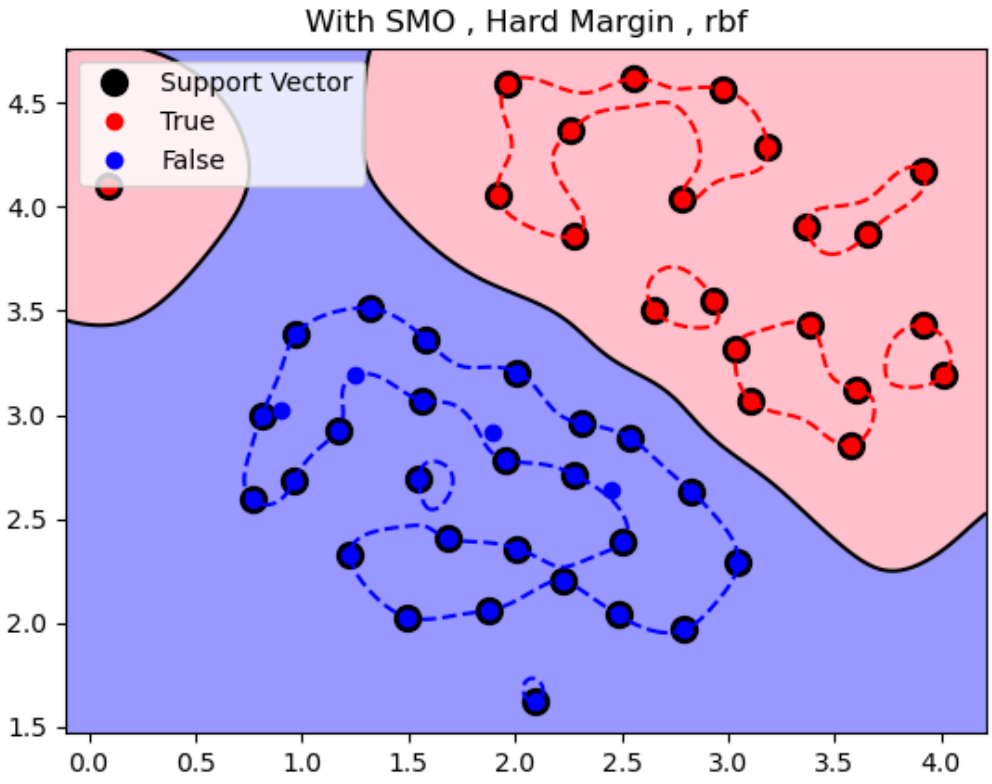
C = 1000:



在实际过程中，发现该处理方式的速度非常的快，和`cvxopt`方法不相上下，并且最终的结果与`cvxopt`效果基本相同，可以说成功的实现了SMO算法来处理SVM。

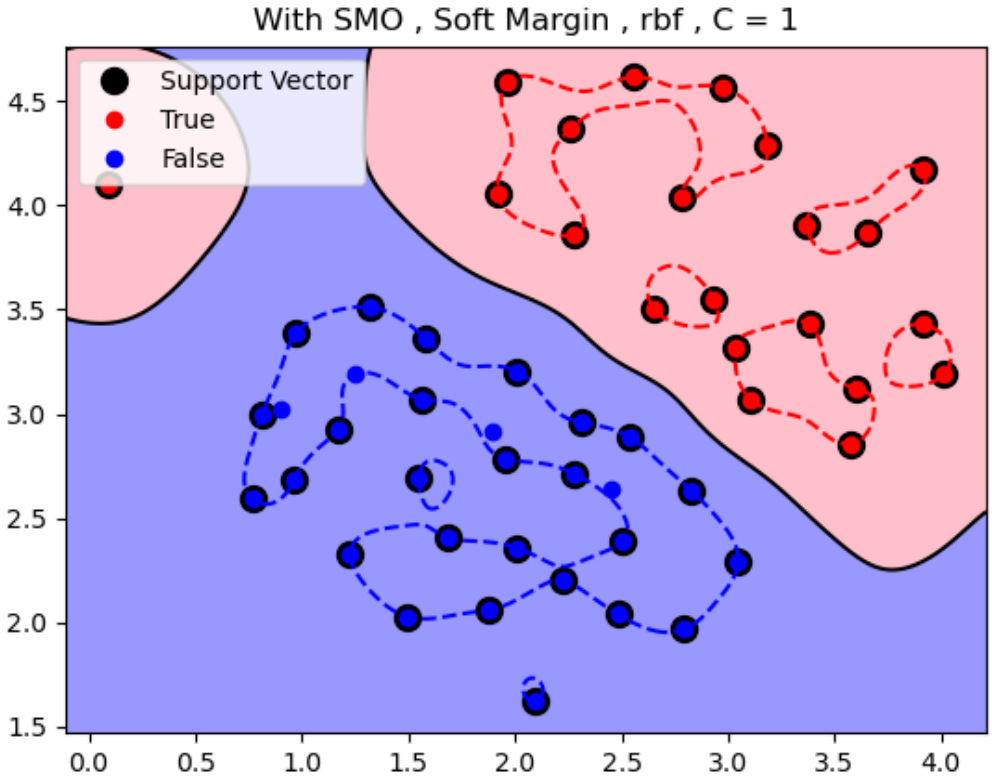
4.6 第一组数据核函数处理

硬间隔：

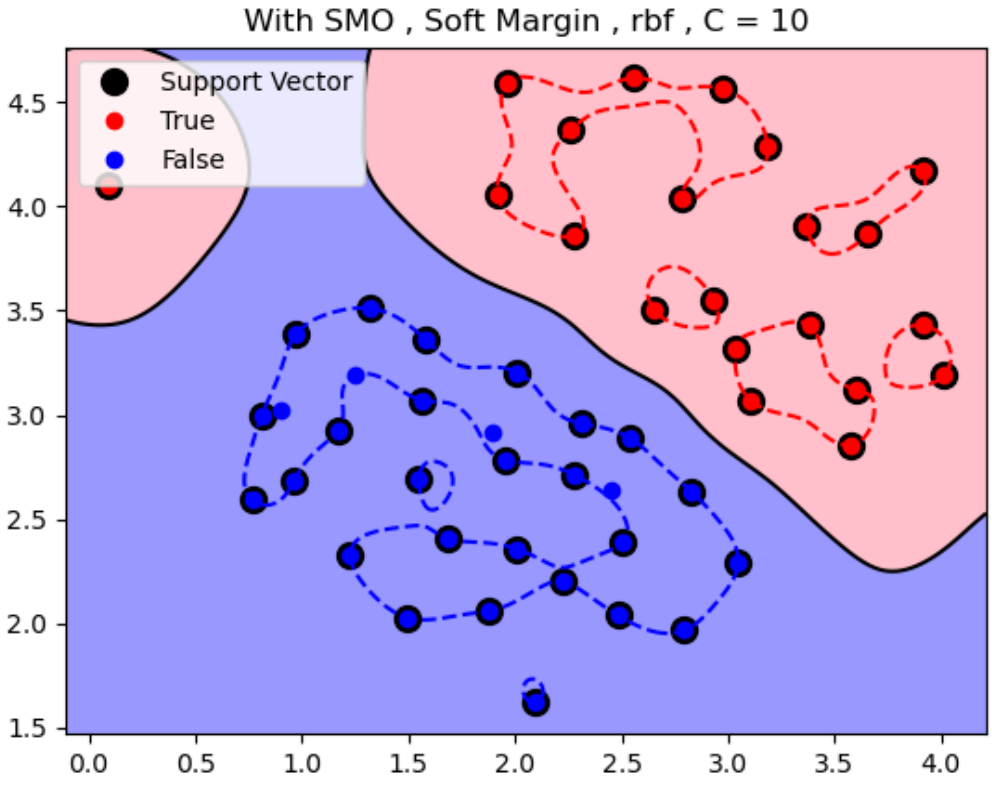


软间隔：

C = 1：



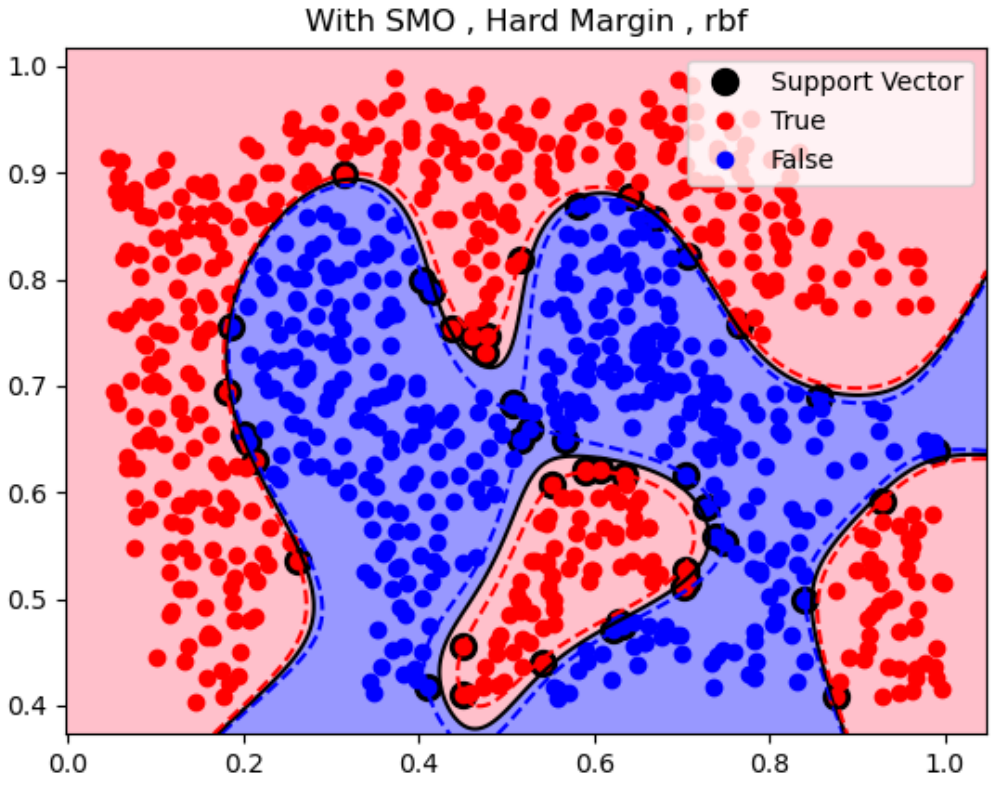
C = 10:



在实际过程中，发现该处理方式的速度非常的快，和`cvxopt`方法不相上下，并且最终的结果与`cvxopt`效果基本相同，可以说成功的实现了SMO算法来处理SVM。

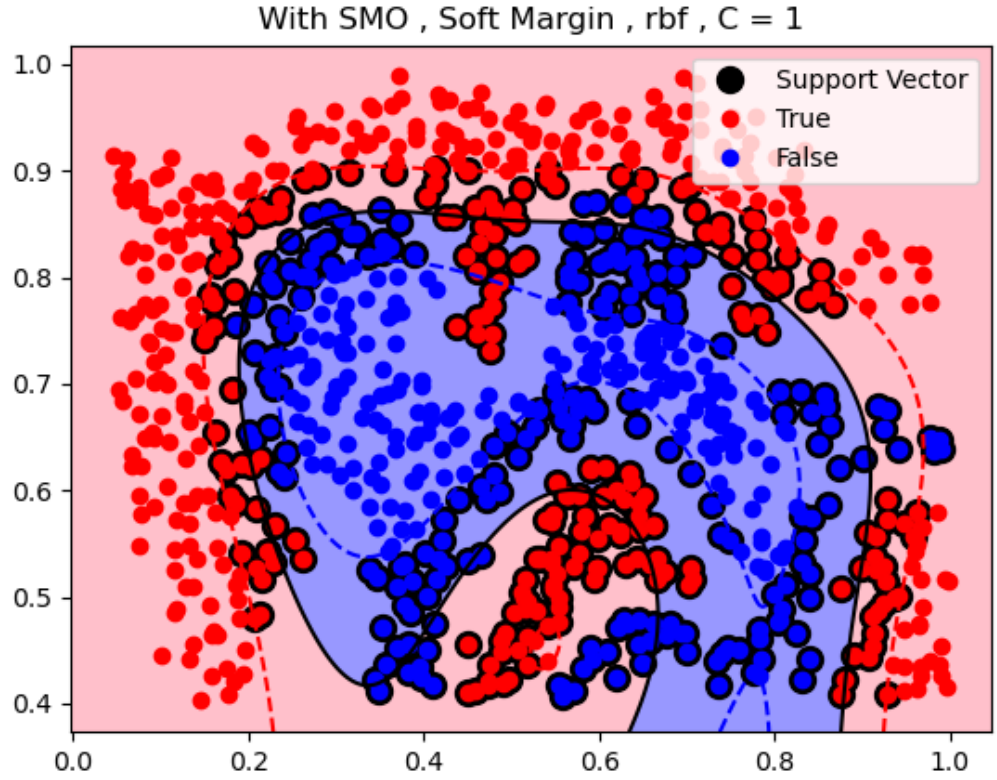
4.7 第二组数据核函数处理

采用硬间隔：此处Hard也是采用设置一个极大的C = 1000来实现。在测试过程中，该过程的计算速度远不如`cvxopt`包的收敛速度。

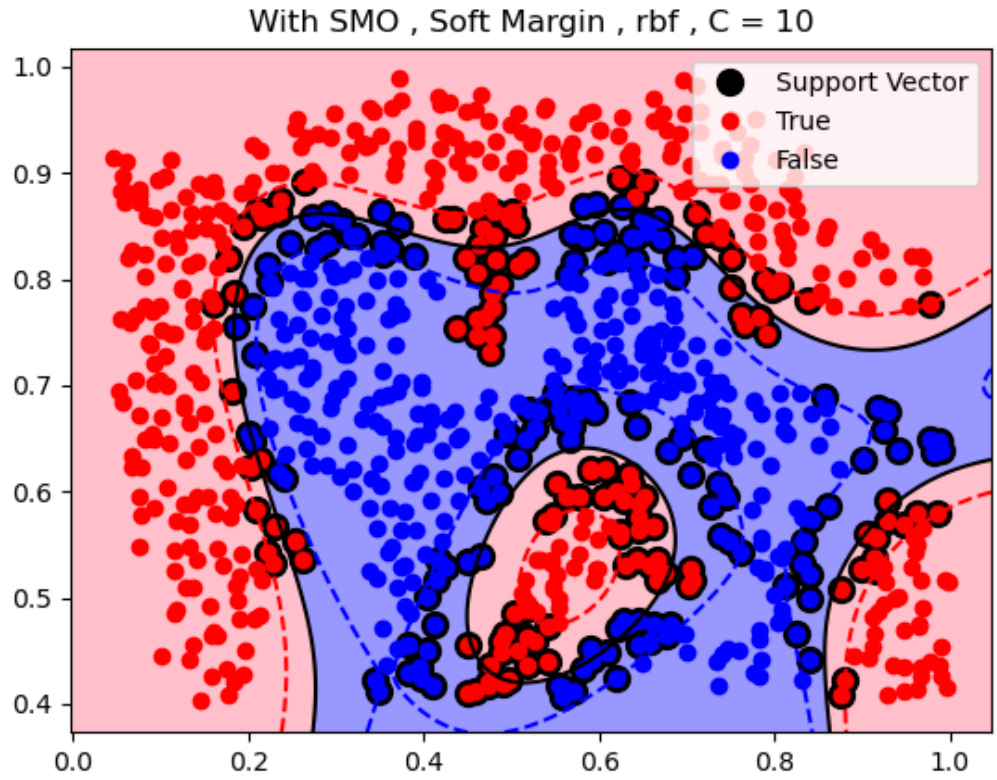


采用软间隔：

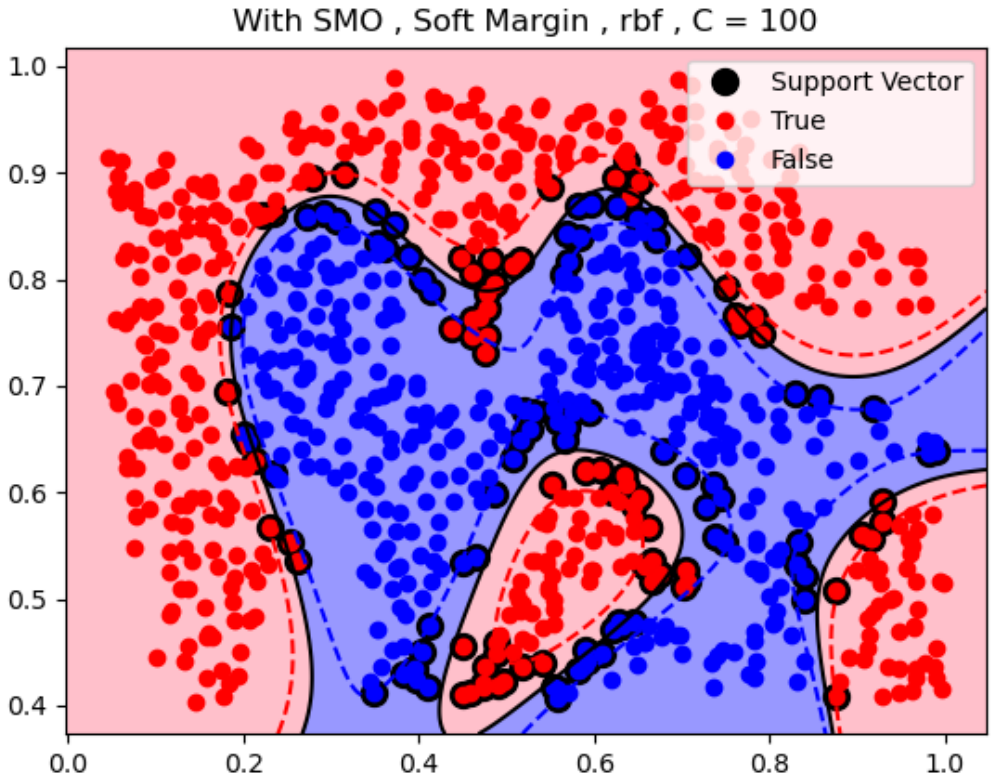
C = 1：


With SMO , Soft Margin , rbf , C = 1
Support Vector
True
False
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.0
0.2
0.4
0.6
0.8
1.0

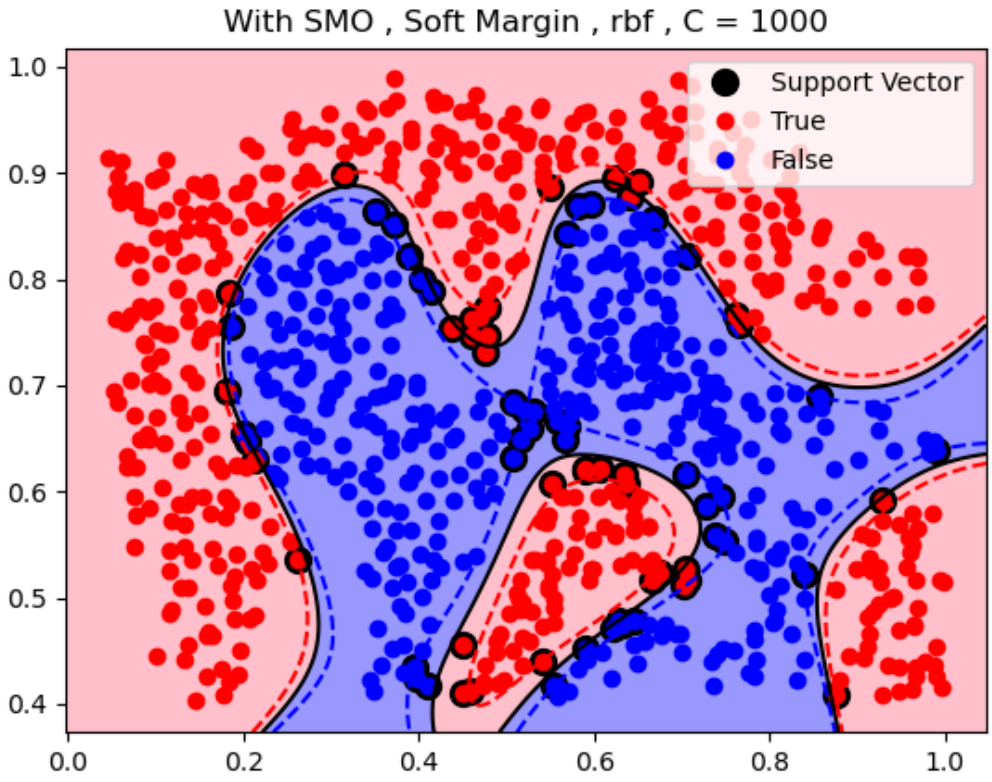
C =10:


With SMO , Soft Margin , rbf , C = 10
Support Vector
True
False
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.0
0.2
0.4
0.6
0.8
1.0

C = 100:



C = 1000:



在实际过程中，发现该处理方式的速度和 `cvxopt` 方法相比，速度很慢，并且最终的结果与 `cvxopt` 效果基本相同，可以说成功的实现了SMO算法来处理SVM。但是还可以通过其他算法来加速该SMO过程。

代码结构

```
/Exp5/
|------- code/
|        |------- SVM.py 实现一个包装较为完善的类，包含多种调用方式
|
|-------- data/
|        |------- ex5data1.mat 线性可分数据集
|        |------- ex5data2.mat 线性不可分数据集
|
|-------- pic/ 存储图像
|
|-------- knowledge.md 学习过程中相关公式Markdown
|
|-------- knowledge.pdf 学习过程中相关公司pdf
|
|-------- 实验报告.md 实验报告Markdown
|
|-------- 实验报告.pdf 实验报告pdf
```

心得体会

在本次实验中，通过实践支持向量机（SVM）算法的不同实现方法以及核函数的应用，我获得了许多宝贵的经验和体会。

理论学习提供了坚实的基础，但在实践中才能真正理解其内涵。在本次实验中，通过编写代码并进行多次实验，我更深刻地理解了SVM算法的工作原理和实现细节。特别是在调试过程中遇到的各种问题，进一步强化了我的问题解决能力。

在实验中，我尝试了不同的惩罚因子C值，并观察了其对分类边界和支持向量数量的影响。实验结果表明，随着C值的增加，分类器变得更加严格，支持向量减少，分类边界更窄。这一过程使我认识到参数选择对模型性能的显著影响，并学会了在不同情况下如何调整参数以获得最佳结果。

通过对比不同实现方法（如梯度下降和SMO算法）的效率和效果，我体会到了算法优化的重要性。虽然不同的方法在最终结果上可能差别不大，但在计算效率和实现难度上却有显著差异。这让我明白了在实际应用中，选择合适的算法不仅能提高效率，还能降低计算资源的消耗。

良好的代码结构和规范对实验的顺利进行至关重要。在本次实验中，我尝试将代码模块化，提高了代码的可读性和可维护性。这一经验使我在今后的编程实践中，将更加注重代码的结构化和规范化。

核函数的引入使得SVM能够处理非线性分类问题。在本次实验中，我应用了径向基函数（RBF）核，并通过调整参数，观察了其对分类效果的影响。这一过程不仅加深了我对核函数的理解，也让我体会到了SVM在处理复杂数据集时的强大能力。

总的来说，本次实验让我在理论知识、实践操作和问题解决能力方面都得到了提升。通过不断的尝试和改进，我不仅掌握了SVM算法的实现和应用，更深刻体会到机器学习领域的丰富和挑战。今后，我将继续努力学习和实践，不断提升自己的专业水平。