基于软间隔SVM算法分类葡萄酒种类

基于软间隔SVM算法分类葡萄酒种类
1 问题描述
2 算法策略
2.1 硬间隔SVM
2.2 软间隔SVM
2.3 SMO算法
2.3.1 SMO算法两个变量的求解过程
2.3.2 SMO算法两个变量的选择过程
2.4 核函数
3 代码实现
3.1 数据集导入与训练集、测试集划分
3.2 特征归一化
3.3 基于SMO算法的软间隔SVM
4 模型表现
4.1 训练轮数对软间隔SVM模型的性能影响
4.2 核函数选择对SVM模型性能的影响
4.3 惩罚参数C的选择对于SVM模型性能的影响
5 总结与展望
参考文献

1 问题描述

葡萄酒是一种广受人们欢迎的含酒精饮品，按照其制作的原材料与制作工艺的不同，可以简单的将其区分为红葡萄酒与白葡萄酒两类。由于葡萄酒的不同风味的不同，不同类别的葡萄酒也适用于不同的场合之中。同时，葡萄酒的特有的风味来源于其中含有的各类化学物质与其物理的属性，而制作原材料的不同选择决定了葡萄酒的理化性质。从而，通过分析葡萄酒中所含有的不同风味物质的含量与其物理特性，我们能够区分葡萄酒的种类，从产品的理化参数，反推其制作的原材料的种类。

因此，在本文中，基于两个葡萄牙北部的葡萄酒质量数据集，我们利用软间隔SVM算法，根据不同种类葡萄酒之间的理化特点，对葡萄酒类型进行分类，并检验不同参数选择的条件下模型的预测准确率。

数据集链接：https://archive.ics.uci.edu/ml/datasets/Wine+Quality (链接到外部网站。)

2 算法策略

2.1 硬间隔SVM

对于给定的一组数据$\{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$，其中$\boldsymbol{x}_i \in \mathbb{R}^d$，$y_i \in \{-1, +1\}$，d为样本的特征数量。对于线性可分的两类样本，则存在有一个超平面$\boldsymbol{w}^\top \boldsymbol{x} + b = 0$将两类样本给间隔分开。通过超平面，即可以得到：

$$y_i \operatorname{sign}\left(w^T x_i + b\right) = 1 \Longleftrightarrow y_i \left(w^T x_i + b\right) > 0$$

考虑到参数(w, b)具有放缩不变性，即对于任意$k > 0$，$y_i(kw^T x_i + kb) > 0$。因此，不妨令$\min |w^T x + b| = 1$，则上式转化为：

$$y_i \left(w^T x_i + b\right) > 1$$

对于SVM模型而言，需要去寻找一个超平面，使其到数据点的间隔最大。而对于间隔，定义为间隔为距离超平面最近的样本到超平面距离的两倍。而对于空间中任意样本点到超平面的距离为$d=\frac{|w^\top x+b|}{\|w\|}$，则得到间隔表达式如下所示：

$$\gamma=2\min\frac{|w^\top x+b|}{\|w\|}=\frac{2}{\|w\|}$$

从而，硬间隔SVM的优化目标即为最大化γ，将间隔函数的表达式等价于凸二次函数，如下所示：

$$\begin{aligned}&\max_{w,b}\frac{2}{\|w\|}\Longrightarrow\min_{w,b}\frac{1}{2}\|w\|^2\\&\text{s.t. }y_i\left(w^\top x_i+b\right)\geq 1,\quad i=1,2,\ldots,m\end{aligned}$$

2.2 软间隔SVM

在2.1中，我们推导了硬间隔SVM的优化目标，然而，使用硬间隔SVM存在有两个问题，分别是不适用于线性不可分数据集并且对离群点敏感。为了解决这些问题，引入了“软间隔”，即允许一些样本点跨越间隔边界甚至是超平面。

于是，为样本量引入惩罚参数C与松弛变量ξ_i，则优化问题转变为：

$$\begin{aligned}&\min_{w,b,\xi}\quad\frac{1}{2}\|w\|^2+C\sum_{i=1}^m\xi_i\\&\text{s.t.}\qquad y_i\left(w^\top x_i+b\right)\geq 1-\xi_i,\quad i=1,2,\ldots,m\\&\qquad\quad\xi_i\geq 0,\quad i=1,2,\ldots m\end{aligned}$$

对于上式中，对于未离群的点，松弛变量都为0；而对于离群点，若点与间隔边界越远，则松弛变量的值也就越大。对于惩罚参数C，当C较大时，对于离群点的惩罚就越大，最后分类时会有较少的点越过间隔边界，模型会变得复杂，反之亦然。

为求解该优化问题，为每条约束映入拉格朗日乘子$\alpha_i\geqslant 0,\beta_i\geqslant 0$，则优化函数的拉格朗日函数即为：

$$\mathcal{L}(w,b,\xi,\alpha,\beta)=\frac{1}{2}\|w\|^2+C\sum_{i=1}^m\xi_i+\sum_{i=1}^m\alpha_i\left[1-\xi_i-y_i\left(w^\top x_i+b\right)\right]+\sum_{i=1}^m\beta_i\left(-\xi_i\right)$$

其对偶问题为：

$$\begin{aligned}&\max_{\alpha,\beta}\min_{w,b,\xi}\mathcal{L}(w,b,\xi,\alpha,\beta)\\&\qquad\text{s.t. }\alpha_i\geq 0,\quad i=1,2,\ldots m\\&\qquad\quad\beta_i\geq 0,\quad i=1,2,\ldots m\end{aligned}$$

对于w,b,ξ参数优化为无约束优化问题，即对其求偏导为0：

$$\begin{aligned}&\frac{\partial\mathcal{L}}{\partial w}=0\Longrightarrow w=\sum_{i=1}^m\alpha_i y_i x_i\\&\frac{\partial\mathcal{L}}{\partial b}=0\Longrightarrow\sum_{i=1}^m\alpha_i y_i=0\\&\frac{\partial\mathcal{L}}{\partial\xi}=0\Longrightarrow\beta_i=C-\alpha_i\end{aligned}$$

将上式带入到拉格朗日函数中，将优化问题简化为：

$$\begin{aligned}&\max_{\alpha}\sum_{i=1}^m\alpha_i-\frac{1}{2}\sum_{i=1}^m\sum_{j=1}^m\alpha_i\alpha_j y_i y_j x_i^\top x_j\\&\text{s.t. }\sum_{i=1}^m\alpha_i y_i=0\\&\qquad 0\leqslant\alpha_i\leqslant C,\quad i=1,2,\ldots m\end{aligned}$$

2.3 SMO算法

如2.2中推导的简化后的软间隔SVM的优化问题，在求出满足条件的最优α 后，即可得 SVM 模型的参数 (w, b) ，进而获得分离超平面。在这个二次规划问题中，有n个变量与n+1个约束条件，当样本容量较大时，该问题无法求解。为此，在求解满足条件的最优α 过程中，选择采用SMO算法，将大问题拆分为一系列小的可解的二次规划问题。SMO算法的实现过程包括有变量求解与变量选择两个部分。

2.3.1 SMO算法两个变量的求解过程

假设选择两个变量α_1与α_2，固定其他变量$\alpha_i, i=3,4...N$，对于2.2中的原始对偶问题可以得到：

$$
\begin{aligned}
&\min_{\alpha} \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K\left(x_i, x_j\right)-\sum_{i=1}^{N} \alpha_i \\
=&\frac{1}{2}\left(\alpha_1 y_1\left(\alpha_1 y_1 K_{11}+\alpha_2 y_2 K_{12}+\sum_{j=3}^{N} \alpha_j y_j K_{1 j}\right)+\alpha_2 y_2\left(\alpha_1 y_1 K_{21}+\alpha_2 y_2 K_{22}+\sum_{j=3}^{N} \alpha_j y_j K_{2 j}\right)\right. \\
&\left.+\sum_{i=3}^{N}\left(\alpha_1 \alpha_i y_{11} K_{1 i}+\alpha_2 \alpha_i y_2 K_{2 i}+\sum_{j=3}^{N} \alpha_i \alpha_j y_i y_j K_{i, j}\right)\right)-\left(\alpha_1+\alpha_2+\sum_{i=3}^{N} \alpha_i\right) \\
=&\frac{1}{2} \alpha_1^2 K_{11}+\frac{1}{2} \alpha_2^2 K_{22}+\alpha_1 \alpha_2 y_1 y_2 K_{12}+\alpha_1 y_1 \sum_{j=3}^{N} \alpha_j y_j K_{1 j}+\alpha_2 y_2 \sum_{j=3}^{N} \alpha_j y_j K_{2 j}-\left(\alpha_1+\alpha_2\right)+\left(\frac{1}{2} \sum_{i=3}^{N} \sum_{j=3}^{N} \alpha_i \alpha_j y_i y_j K_{i, j}-\sum_{i=3}^{N} \alpha_i\right)
\end{aligned}
$$

其中，$K_{ij}=K\left(x_i, x_j\right)=K_{ji}=K\left(x_j, x_i\right)$。

令$M=\frac{1}{2} \sum_{i=3}^{N} \sum_{j=3}^{N} \alpha_i \alpha_j y_i y_j K_{i, j}-\sum_{i=3}^{N} \alpha_i$，此时的优化问题更新为：

$$
\begin{gathered}
\min_{\alpha_1, \alpha_2} \frac{1}{2} \alpha_1^2 K_{11}+\frac{1}{2} \alpha_2^2 K_{22}+\alpha_1 \alpha_2 y_1 y_2 K_{12}+\alpha_1 y_1 \sum_{j=3}^{N} \alpha_j y_j K_{1 j}+\alpha_2 y_2 \sum_{j=3}^{N} \alpha_j y_j K_{2 j}-\left(\alpha_1+\alpha_2\right)+M \\
s.t. \alpha_1 y_1+\alpha_2 y_2=-\sum_{i=3}^{N} \alpha_i=M_2 \\
0 \leq \alpha_i \leq C, \quad i=1,2
\end{gathered}
$$

接下来，根据约束条件，可以知道(α_1, α_2)位于区域$[0, C] * [C, 0]$的对角线的直线上，故目标函数需要求解的时，即为在一条平行于对角线的线段上的最优值。假设问题的初始解为(α_1, α_2)，最优解为(α_1^*, α_2^*)，不考虑约束条件的最优解为$(\alpha_1^{**}, \alpha_2^{**})$。

对于不考虑约束条件的最优解为$(\alpha_1^{**}, \alpha_2^{**})$的求解，记输入的$x_i$的预测值与真实值之差为$E_i$，则：

$$
E_i=pred_i-y_i=\left(\sum_{j=1}^{N} \alpha_j y_j K\left(x_i, x_j\right)+b\right)-y_i \quad i=1,2
$$

可以得到：

$$
\alpha_2^{**}=\alpha_2+\frac{y_2-\left(E_1-E_2\right)}{\eta} \quad \eta=K_{11}+K_{22}-2 K_{12}
$$

由于α_2^*存在约束条件，故$L \leq \alpha_2^* \leq H$，其中，L与H为端点的边界，故：

$$
\begin{gathered}
L=\max \left(0, \alpha_2-\alpha_1\right), \quad H=\min \left(C, C+\alpha_2-\alpha_1\right) \quad y_1 ! = y2 \\
L=\max \left(0, \alpha_2^{+} \alpha_1-C\right), \quad H=\min \left(C, \alpha_2+\alpha_1\right) \quad y_1=y2
\end{gathered}
$$

总结上述推导过程，得到α_2^*的解为：

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

在计算得到α_2^*后，可以计算出α_1^*的值，表示如下：

$$\alpha_1^* = \alpha_2 + y_1 y_2 \left(\alpha_2 - \alpha_2^*\right)$$

2.3.2 SMO算法两个变量的选择过程

1.第一个变量的选择

对于第一个变量，首先检验样本点(x_i, y_i)是否符合KKT条件，即：

$$\begin{aligned}
&\alpha_i = 0, <=> y_i g\left(x_i\right) \geq 1 \\
&0 < \alpha < C, <=> y_i g\left(x_i\right) = 1 \\
&\alpha_i = C, <=> y_i g\left(x_i\right) \leqslant 1
\end{aligned}$$

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

在检验过程后，循环首先便利所有在间隔边界上的支持向量点，检验其是否满足KKT条件，在训练样本中选取违反KKT条件最严重的样本点对应的变量作为第一个变量。

2.第二个变量的选择

假设在第一步中找到了第一个变量α_1，则对于第二个变量α_2，由于最优化的α_2^*与$|E_1 - E_2|$相关，因此需要使得对应的$|E_1 - E_2|$最大。在选择α_2的过程中，首先遍历在间隔边界上的点，将其作为α_2使用；若目标函数没有足够的下降，则遍历整个数据集；若仍没有找到合适的α_2，则放弃α_1，回到第一步重新寻找合适的变量。

3.更新阈值b与误差E_i

在完成两个变量的选择之后，需要更新阈值b与误差E_i。对于阈值b，由KKT条件$\sum_{i=1}^N \alpha_i y_i K_{i1} + b = y_1$可知，

$$b_1^* = y_1 - \sum_{i=3}^N \alpha_i y_i K_{i1} - \alpha_1^* y_1 K_{11} - \alpha_2^* y_2 K_{21}$$

又因为$E_1 = \sum_{i=3}^N \alpha_i y_i K_{i1} + \alpha_1 y_1 K_{11} + \alpha_2 y_2 K_{21} + b - y_1$，得到：

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

同理可得：

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

从而，如上面的公式推导所示，b的值更新为：

$$\begin{aligned}
&b^* = b_1^* = b_2^* \quad 0 < \alpha_i^* < C \\
&b^* = \left(b_1^* + b_2^*\right)/2 \quad \alpha_i^* = 0 or C
\end{aligned}$$

在得到b的更新之后，更新E_i的值：

$$E_i^* = \sum_s y_j \alpha_j K\left(x_i, x_j\right) + b^* - y_i$$

2.4 核函数

在2.1与2.2中的硬间隔与软间隔的SVM都只适用于线性可分的数据集，则对于线性不可分的数据集而言，则需要使用核函数，通过一个映射$Phi: \mathbb{R}^d \longrightarrow \mathbb{R}^{\tilde{d}}$将数据在新的特征空间中线性可分。则对于一个核函数$\mathcal{K}\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \phi(\boldsymbol{x}_i)^{\top} \phi\left(\boldsymbol{x}_j\right)$，则优化问题变为：

$$\max_{\alpha} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{i=1}^{m} \alpha_i \alpha_j y_i y_j \kappa\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right)$$
$$\text{s.t.} \sum_{i=1}^{m} \alpha_i y_i = 0$$
$$0 \leqslant \alpha_i \leqslant C, \quad i = 1, 2, \ldots m$$

对于要求解的超平面变为：

$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{w}^{\top} \phi(\boldsymbol{x}) + b \\ &= \sum_{i \in SV} \alpha_i y_i \phi\left(\boldsymbol{x}_i\right) \phi(\boldsymbol{x}) + b \\ &= \sum_{i \in SV} \alpha_i y_i \kappa\left(\boldsymbol{x}_i, \boldsymbol{x}\right) + b \end{aligned}$$

其中 SV代表所有支持向量的集合。这样我们就能通过在低维空间计算核函数的方式，以较小的计算量解决非线性分类问题。

3 代码实现

3.1 数据集导入与训练集、测试集划分

利用$pandas$库中的文件操作函数，读入数据集。之后，以80%训练集、20%测试集的形式随机从读取的数据集中随机划分，得到实验中使用的训练集与测试集。

```
# 导入数据集并拼接为一个data_frame，颜色标签：red=1，white=-1，并随即生成训练集与测试集
def load_data(df1, df2):
    df_red = pd.read_csv(df1, sep=";")
    df_red.columns = ["fixed acidity", "volatile acidity", "citric acid",
"residual sugar", "chlorides",
                      "free sulfur dioxide", "total sulfur dioxide", "density",
"pH", "sulphates", "alcohol", "quality"]
    df_red["type"] = 1

    df_white = pd.read_csv(df2, sep=";")
    df_white.columns = ["fixed acidity", "volatile acidity", "citric acid",
"residual sugar", "chlorides",
                        "free sulfur dioxide", "total sulfur dioxide",
"density", "pH", "sulphates", "alcohol",
                        "quality"]
    df_white["type"] = -1

    df_red = np.array(df_red)
    df_white = np.array(df_white)

    df_red_train = df_red[:int(df_red.shape[0] * 0.8), :]
    df_red_test = df_red[int(df_red.shape[0] * 0.8) + 1:-1, :]
    df_white_train = df_white[:int(df_white.shape[0] * 0.8), :]
    df_white_test = df_white[int(df_white.shape[0] * 0.8) + 1:-1, :]

    train_set = np.concatenate((df_red_train, df_white_train))
    test_set = np.concatenate((df_red_test, df_white_test))

    return train_set, test_set

# 导入数据,进行分类问题处理
red_wine = r".\winequality-red.csv"
```

```
white_wine = r".\winequality-white.csv"

train_data, test_data = load_data(red_wine, white_wine)

tr_data = train_data[:, :train_data.shape[1]-2]
tr_label = train_data[:, train_data.shape[1]-1:]
te_data = test_data[:, :test_data.shape[1]-2]
te_label = test_data[:, test_data.shape[1]-1:]
```

3.2 特征归一化

对导入数据中的特征参数归一化，代码实现如下所示：

```
# 特征归一化
mu = []
std = []


def normalize(data):
    for i in range(0, data.shape[1] - 2):
        data[:, i] = ((data[:, i] - np.mean(data[:, i])) / np.std(data[:, i]))
        mu.append(np.mean(data[:, i]))
        std.append(np.std(data[:, i]))
```

3.3 基于SMO算法的软间隔SVM

```
# 定义SVM类
class SVM(object):
    """
    参数解释：
    data: 训练的数据集
    label: 标签
    C: 惩罚系数
    b: bias
    kernel：选择的核函数与核函数初始化参数
    alpha: 拉格朗日乘子
    error: 误差值
    kernel: 核函数的值
    """

    def __init__(self, data, label, epoch=500, C=0.5, kernel_opt=("linear",
0.5)):
        self.data = data
        self.label = label
        self.C = C
        self.b = 0
        self.epoch = epoch
        self.kernel_opt = kernel_opt
        num_data = self.data.shape[0]
        self.alpha = np.zeros((num_data, 1))
        self.error = np.zeros((num_data, 2))
        self.kernel = None

    """
    初始化核函数：可用核函数包括线性核函数，高斯核函数和多项式核函数
    init_kernel_value：返回各个样本之间的核函数值(value_shape = num_data*1)
```

```
    init_kernel：返回样本总体的核函数值(value_shape = num_data*num_data)
    """

    def init_kernel_value(self, temp_data):
        num_data = self.data.shape[0]
        kernel_init_value = self.kernel_opt[1]
        kernel_i = np.zeros((num_data, 1))
        if self.kernel_opt[0] == "linear":
            kernel_i = np.dot(self.data, temp_data.T)
        elif self.kernel_opt[0] == "rbf":
            for i in range(num_data):
                temp_diff = self.data[i, :] - temp_data
                kernel_i[i] = np.exp(np.dot(temp_diff, temp_diff.T) / (-2.0 *
kernel_init_value ** 2))
            kernel_i = np.squeeze(kernel_i)
        elif self.kernel_opt[0] == "polynomial":
            for i in range(num_data):
                kernel_i[i] = (np.dot(self.data[i, :], temp_data.T) + 1) **
kernel_init_value
            kernel_i = np.squeeze(kernel_i)
        else:
            print("Available kernel choice: linear, rbf and polynomial")
        return kernel_i

    def init_kernel(self):
        num_data = self.data.shape[0]
        kernel = np.zeros((num_data, num_data))
        for i in range(num_data):
            kernel[:, i] = self.init_kernel_value(self.data[i, :])
        return kernel

    """
    cal_error：计算第i个alpha对应误差值
    choose_second_alpha：选择第二个变量并返回第二个变量的误差
    update_error：计算更新误差值
    choose_update_alpha：选择并更新两个alpha, b, error
    """

    def cal_error(self, i_index):
        temp_prediction = float(np.dot(np.multiply(self.alpha, self.label).T,
self.kernel[:, i_index]) + self.b)
        temp_error = temp_prediction - float(self.label[i_index])
        return temp_error

    def choose_second_alpha(self, a_index, a_error):
        self.error[a_index] = [1, a_error]
        choice_list = np.nonzero(self.error[:, 0])[0]
        best_gap = 0
        b_error = 0
        b_index = -1
        # 若有多个备选的变量
        if len(choice_list) > 1:
            for temp_index in choice_list:
                if temp_index == a_index:
                    continue
                temp_error = self.cal_error(temp_index)
                if abs(temp_error - a_error) > best_gap:
                    best_gap = abs(temp_error - a_error)
```

```
                    b_index = temp_index
                    b_error = temp_error
        # 若只有一个或没有备选变量，则随机选择第二个变量
        else:
            b_index = a_index
            while b_index == a_index:
                b_index = np.random.randint(self.data.shape[0])
            b_error = self.cal_error(b_index)
        return b_index, b_error

    def update_error(self, i):
        temp_error = self.cal_error(i)
        self.error[i] = [i, temp_error]

    def choose_update_alpha(self, a_index):
        a_error = self.cal_error(a_index)
        if (self.alpha[a_index] < self.C) or (self.alpha[a_index] > 0):
            b_index, b_error = self.choose_second_alpha(a_index, a_error)
            last_a_alpha = self.alpha[a_index].copy()
            last_b_alpha = self.alpha[b_index].copy()
            # 计算上下界
            if self.label[a_index] != self.label[b_index]:
                l = max(0, self.alpha[b_index] - self.alpha[a_index])
                h = min(self.C, self.C + self.alpha[b_index] -
self.alpha[a_index])
            else:
                l = max(0, self.alpha[b_index] + self.alpha[a_index] - self.C)
                h = min(self.C, self.alpha[b_index] + self.alpha[a_index])
            if l == h:
                return 0
            # 计算eta值
            eta = self.kernel[a_index, a_index] + self.kernel[b_index, b_index]
- 2.0*self.kernel[a_index, b_index]
            if eta <= 0:
                return 0
            # 更新第二个alpha，根据范围约束最终的alpha(b)
            self.alpha[b_index] += self.label[b_index] * (a_error - b_error) /
eta
            if self.alpha[b_index] > h:
                self.alpha[b_index] = h
            if self.alpha[b_index] < l:
                self.alpha[b_index] = l
            # 若第二个alpha值不在改变，则结束
            if abs(last_b_alpha - self.alpha[b_index]) < 1e-5:
                self.update_error(b_index)
                return 0
            # 更新第一个alpha值
            self.alpha[a_index] += self.label[a_index] * self.label[b_index] *
(last_b_alpha - self.alpha[b_index])
            # 更新b
            b1 = self.b - a_error - self.label[a_index] * self.kernel[a_index,
a_index] * (self.alpha[a_index] - last_a_alpha) - self.label[b_index] *
self.kernel[a_index, b_index] * (self.alpha[b_index] - last_b_alpha)
            b2 = self.b - b_error - self.label[a_index] * self.kernel[a_index,
b_index] * (self.alpha[a_index] - last_a_alpha) - self.label[b_index] *
self.kernel[b_index, b_index] * (self.alpha[b_index] - last_b_alpha)
            if 0 < self.alpha[a_index] < self.C:
                self.b = b1
```

```
            elif 0 < self.alpha[b_index] < self.C:
                self.b = b2
            else:
                self.b = (b1 + b2) / 2.0
            # 更新error
            self.update_error(a_index)
            self.update_error(b_index)
            return 1
        else:
            return 0

    def train(self):
        all_dataset = True
        self.kernel = self.init_kernel()
        num_data = self.data.shape[0]
        pair_change = 0
        iteration = 0
        while (iteration < self.epoch) and (pair_change > 0 or all_dataset):
            print("\t epoch", iteration)
            pair_change = 0
            if all_dataset:
                for i in range(num_data):
                    print(i)
                    pair_change += self.choose_update_alpha(i)
                iteration += 1
            else:
                boundary_item = []
                for i in range(num_data):
                    if 0 < self.alpha[i, 0] < self.C:
                        boundary_item.append(i)
                for j in boundary_item:
                    pair_change += self.choose_update_alpha(j)
                iteration += 1
            if all_dataset:
                all_dataset = False
            elif pair_change == 0:
                all_dataset = True
        return self

    def predict(self, test_data_i):
        kernel_value = self.init_kernel_value(test_data_i)
        predict = np.dot(np.multiply(self.label, self.alpha).T, kernel_value) +
self.b
        return predict

    def cal_acc(self, test_data, test_label):
        num_data = test_data.shape[0]
        count = 0.0
        for i in range(num_data):
            temp_pred = self.predict(test_data[i, :])
            if np.sign(temp_pred) == np.sign(test_label[i]):
                count += 1
        accuracy = 1.0 * count / num_data
        return accuracy

normalize(tr_data)
normalize(te_data)
demo = SVM(tr_data, tr_label, kernel_opt=("rbf", 0.6))
```

```
demo_train = demo.train()
acc = demo.cal_acc(tr_data, tr_label)
print(acc)
```

4 模型表现

在葡萄酒类型分类任务中，我们选择了固定酸度，挥发性酸度，柠檬酸，残糖，氯化物，游离二氧化硫，总二氧化硫，密度，pH，硫酸盐含量与酒精含量这11种理化参数作为特征数据，以红、白两种葡萄酒类型作为分类标准，以（-1，1）作为分类标签，希望通过以上的理化特征，利用基于SMO算法的软间隔SVM模型完成对葡萄酒种类的区分。

4.1 训练轮数对软间隔SVM模型的性能影响

在模型实验中，选择参数为$C=0.5$，核函数为$(" \ rbf \ ", 0.6)$，训练轮数为12轮，间隔数为2随着模型的训练轮数的增加，模型的准确率也随之上升。当训练时间达到10轮时，模型预测的准确率不在发生显著变化，模型收敛。同时，可以从图中读出，当参数为$C=0.5$，核函数为$(" \ rbf \ ", 0.6)$时，软间隔SVM模型的分类准确性为**0.914**。考虑到二分类问题的模型分类准确率基线为0.5，而SVM模型的分类准确性接近于1，这一结果表明SVM模型能够有效的基于葡萄酒的各类理化特征参数，对葡萄酒的种类进行区分，也体现出SVM模型对二分类任务的有效性与准确性。

4.2 核函数选择对SVM模型性能的影响

在2.4中介绍了核函数通过将原始数据映射到另一个特征空间以实现数据非线性分割，而考虑到不同核函数所具有的不同函数特点，在这一部分中，对核函数的选择进行实验，以分析核函数选择对SVM模型性能的影响。在实验中，选择了3种常用的核函数：线性核、多项式核与RBF核，分别以固定的训练轮数观察SVM模型的分类准确性。同时，记录分析使用不同核函数的模型训练时间，从运行速度的层面分析模型的性能。实验结果如下表所示：

表一 SVM模型性能与核函数选择的关系

	线性核	**多项式核**	**RBF核**
分类准确性	0.869	0.754	**0.914**
运行时间（s/轮）	**290.0**	349.7	423.7

如上表所示，在三种核函数中，使用RBF核的SVM模型的分类准确性最高，达到了**0.914**；而线性核的运行时间最短，为**290s**一轮。这样的实验结果体现出RBF核拟合能力强且预测准确率高的优点与线性核运行速度快的特点。

4.3 惩罚参数C的选择对于SVM模型性能的影响

在2.2中介绍软间隔SVM与硬间隔SVM之间的显著不同就是松弛变量与惩罚参数C。为了检验惩罚参数C的选择对于SVM模型性能的影响，在这一部分中，我们选取选择核函数为$(rbf, 0.6)$与不同的惩罚参数（0.05, 0.1, 0.3, 0.5, 2, 5）来观察SVM模型性能的变化。

如上图所示，可以看出，随着惩罚参数C的增大，SVM模型分类的准确率下降，模型的性能下降。对于这一现象，是由于当C较大时，对于离群点的惩罚就越大，最后分类时会有较少的点越过间隔边界，模型会变得复杂，反之亦然。因此，当C较大时，模型复杂度高，容易发生过拟合的情况，导致在测试集上的性能表现有所下降。

5 总结与展望

在本文中，以两个葡萄酒理化性质数据集为基础，我们利用基于SMO算法的软间隔SVM模型，通过对葡萄酒理化性质的特征参数的运算，对葡萄酒的类型进行分类。通过对模型的实验，可以得出以下结论：

- 对于基于SMO算法的软间隔SVM模型，随着模型的训练轮数的增加，模型的准确率也随之上升，模型分类准确性为**0.914**。这一结果表明SVM模型能够有效的基于葡萄酒的各类理化特征参数，对葡萄酒的种类进行区分，也体现出SVM模型对二分类任务的有效性与准确性。
- 对于核函数的选择问题，在常见的核函数中，可以得到：（1）使用RBF核的SVM模型的分类准确性最高，达到了**0.914**，具有拟合能力强且预测准确率高的优点；（2）线性核的运行时间最短，为**290s**一轮，具有运行速度快的特点。
- 对于惩罚参数C，随着惩罚参数C的增大，SVM模型分类的准确率下降，模型的性能下降。这是由于当C较大时，模型复杂度高，容易发生过拟合的情况，导致在测试集上的性能表现有所下降。

参考文献

[1]P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis.
Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553, 2009.