菊安酱的机器学习第5期

菊安酱的直播间：https://live.bilibili.com/14988341

每周一晚8:00 菊安酱和你不见不散哦~(^o^)/~

更新日期：2018-12-3

作者：菊安酱

课件内容说明：

- 本文为作者参考众多书籍和博客所写，转载请注明作者和出处
- 如果想获得此课件及录播视频，可扫描左边二维码，回复"k"进群
- 如果想获得2小时完整版视频，可扫描右边二维码或点击如下链接
- 若有任何疑问，请给作者留言。

交流群二维码

完整版视频及课件

直播视频及课件：http://www.peixun.net/view/1278.html

完整版视频及课件：http://edu.cda.cn/course/966

12期完整版课纲

直播时间：每周一晚8:00

直播内容：

时间	期数	算法
2018/11/05	第1期	k-近邻算法
2018/11/12	第2期	决策树
2018/11/19	第3期	朴素贝叶斯
2018/11/26	第4期	Logistic回归
2018/12/03	第5期	支持向量机
2018/12/10	第6期	AdaBoost 算法
2018/12/17	第7期	线性回归
2018/12/24	第8期	树回归
2018/12/31	第9期	K-均值聚类算法
2019/01/07	第10期	Apriori 算法
2019/01/14	第11期	FP-growth 算法
2019/01/21	第12期	奇异值分解SVD

支持向量机

一、什么是SVM?

支持向量机（Support Vector Machine,SVM）是用于分类的一种算法，也属于有监督学习的范畴。

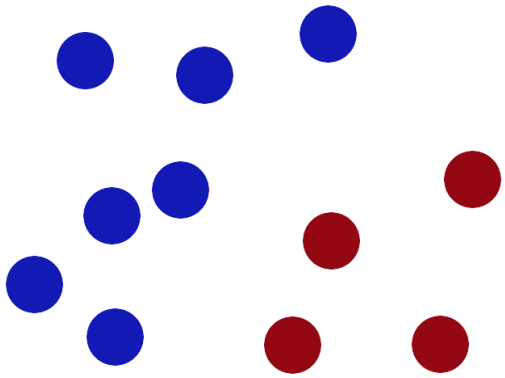
让我们先从一个大侠与反派的故事开始吧~

【该故事来源于https://www.reddit.com上的一个话题讨论：让5岁小孩也能看懂的SVM】

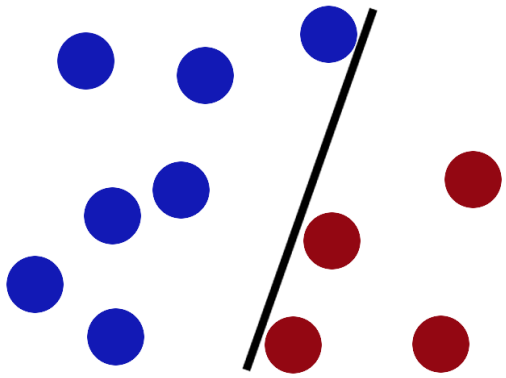
在很久以前，大侠的心上人被反派囚禁，大侠想要去救出他的心上人，于是便去和反派谈判。反派说只要你能顺利通过三关，我就放了你的心上人。

现在大侠的闯关正式开始：

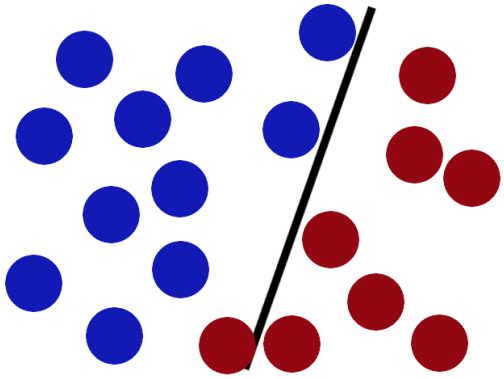
第一关：反派在桌子上似乎有规律地放了两种颜色的球，说：你用一根棍子分离开他们，要求是尽量再放更多的球之后，仍然适用。



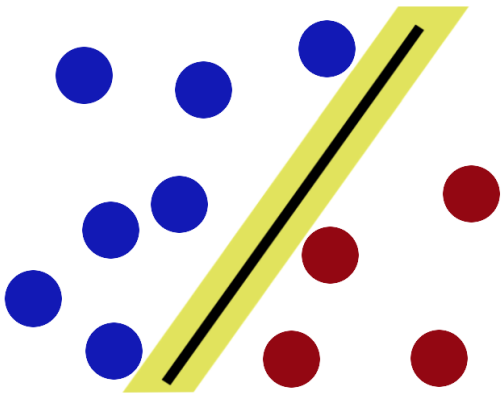
大侠很干净利索的放了一根棍子如下：



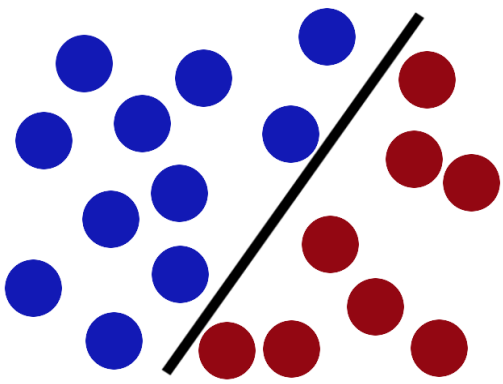
第二关：反派在桌子放上了更多的球，似乎有一个红球站错了阵营。



SVM就是试图把棍放在最佳位置，好让在棍的两边有尽可能大的间隙。

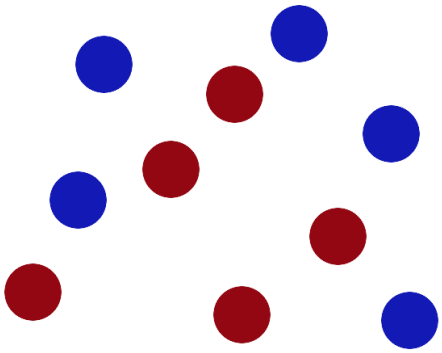


于是大侠将棍子调整如下，现在即使反派放入更多的球，棍子仍然是一个很好的分界线。

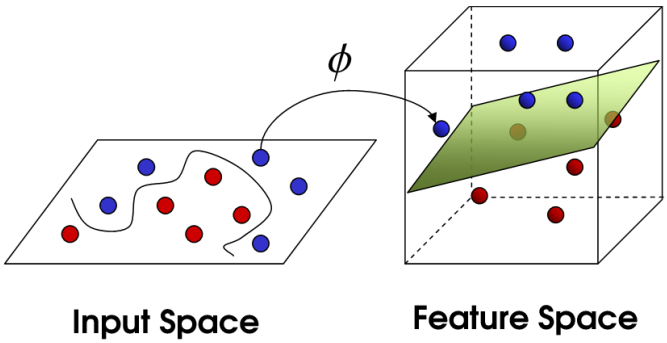


其实在SVM工具箱里还有另一个更加重要的**trick**。反派看到大侠已经学会了一个trick，于是心生一计，给大侠更难的一个挑战。

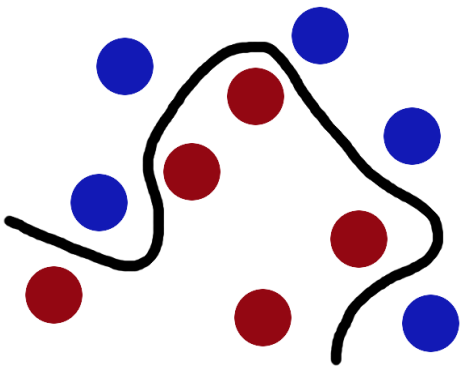
第三关：反派将球散乱地放在桌子上。



现在大侠已经没有方法用一根棍子将这些球分开了，怎么办呢？大侠灵机一动，使出三成内力拍向桌子，然后桌子上的球就被震到空中，说时迟那时快，大侠瞬间抓起一张纸，插到了两种球的中间。



现在从反派的角度看这些球，这些球像是被一条曲线分开了。于是反派乖乖地放了大侠的心上人。

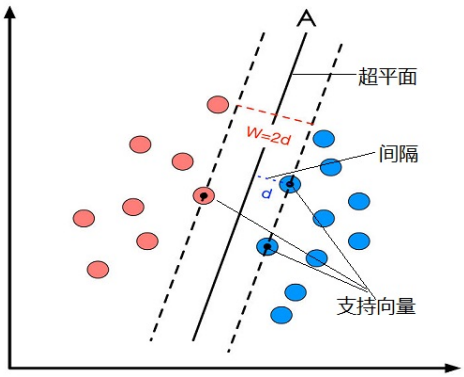


从此之后，江湖人便给这些分别起了名字，把这些球叫做 **「data」**，把棍子叫做 **「classifier」**，最大间隙trick 叫做**「optimization」**， 拍桌子叫做**「kernelling」**，那张纸叫做**「hyperplane」**。

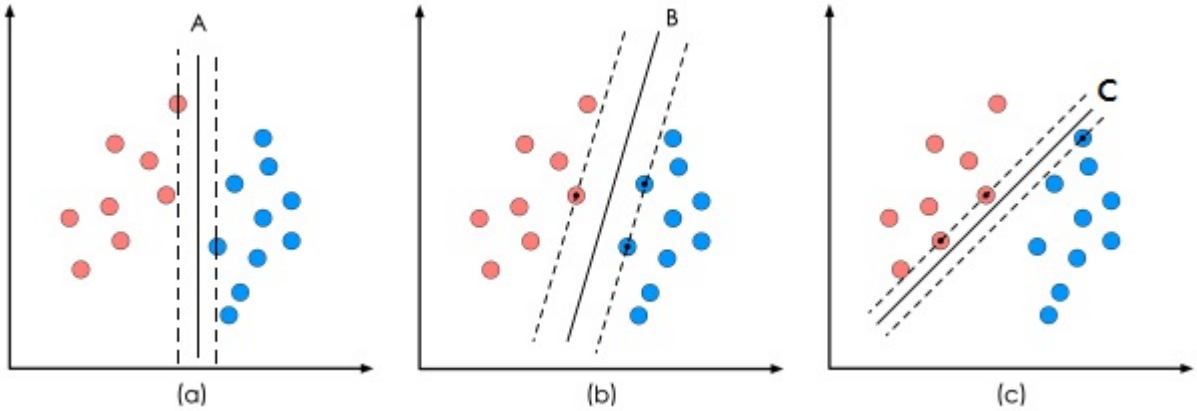
更为直观地感受一下吧（需翻墙）：https://www.youtube.com/watch?v=-Z4aojJ-pdg

概述一下：

当一个分类问题，数据是**线性可分(linearly separable)**的，也就是用一根棍就可以将两种小球分开的时候，我们只要将棍的位置放在让小球距离棍的距离最大化的位置即可，寻找这个最大间隔的过程，就叫做**最优化**。但是，现实往往是很残酷的，一般的数据是线性不可分的，也就是找不到一个棍将两种小球很好的分类。这个时候，我们就需要像大侠一样，将小球拍起，用一张纸代替小棍将小球进行分类。想要让数据飞起，我们需要的东西就是**核函数(kernel)**，用于切分小球的纸，就是**超平面**(hyperplane)。如果数据集是N维的，那么超平面就是N-1维的。



把一个数据集正确分开的超平面可能有多个（如下图），而那个具有“最大间隔”的超平面就是SVM要寻找的最优解。而这个真正的最优解对应的两侧虚线所穿过的样本点，就是SVM中的支持样本点，称为“**支持向量(support vector)**”。支持向量到超平面的距离被称为**间隔(margin)**。



维基百科对SVM的介绍(需翻墙)：

https://zh.wikipedia.org/wiki/%E6%94%AF%E6%8C%81%E5%90%91%E9%87%8F%E6%9C%BA

二、线性SVM

一个最优化问题通常有两个最基本的因素：

1）目标函数，也就是你希望什么东西的什么指标达到最好；

2）优化对象，你期望通过改变哪些因素来使你的目标函数达到最优。

在线性SVM算法中，目标函数显然就是那个“间隔”，而优化对象则是超平面。

我们以线性可分的二分类问题为例。

1. 超平面方程

在线性可分的二分类问题中，超平面其实就是一条直线。相信直线方程大家都不陌生：

$y = ax + b$ (公式1)

现在我们做个小小的改变，让原来的$\boldsymbol{x}$轴变成$\boldsymbol{x_1}$轴，$\boldsymbol{y}$变成$\boldsymbol{x_2}$轴，于是公式(1)中的直线方程会变成下面的样子:

$x_2 = ax_1 + b$ (公式2)

$ax_1 + (-1)x_2 + b = 0$ （公式3)

向量形式可以写成：

$[a, -1]\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b = 0$ (公式4)

进一步可表示为：

$\boldsymbol{\omega}^T\boldsymbol{x} + b = 0$ (公式5)

看到变量$\boldsymbol{\omega}, \boldsymbol{x}$略显粗壮的身体了吗？他们是黑体，表示变量是个向量，$\boldsymbol{\omega} = [\omega_1, \omega_2]^T$，$\boldsymbol{x} = [x_1, x_2]^T$。一般我们提到向量的时候，都默认是列向量，所以对$\boldsymbol{\omega}$进行了转置。这里向量$\boldsymbol{\omega}$与直线是相互垂直的（感兴趣的小伙伴可以推导一下），也就是说$\boldsymbol{\omega}$控制了直线的方向，b就是截距，它控制了直线的位置。

2. 间隔的计算公式

“间隔”其实就是点到直线的距离，如果你在百度文库里面搜索 “点到直线距离推导公式”，那么你会得到至少6、7种推导方法。这里采用向量法：

$$d = \frac{|\boldsymbol{\omega}^T\boldsymbol{x} + b|}{||\boldsymbol{\omega}||} \qquad (\text{公式}6)$$

这里$||\boldsymbol{\omega}||$是向量$\boldsymbol{\omega}$的模，假如$\boldsymbol{\omega} = [\omega_1, \omega_2]^T$，则$||\boldsymbol{\omega}|| = \sqrt{\omega_1^2 + \omega_2^2}$，表示在空间中向量的长度，$\boldsymbol{x} = [x_1, x_2]^T$就是支持向量样本点的坐标。$\boldsymbol{\omega}, b$就是超平面方程的参数。

我们的目标是找出一个分类效果好的超平面作为分类器。分类器的好坏评定依据是分类间隔的$W = 2d$的大小，即分类间隔W越大，我们认为这个超平面的分类效果越好。而追求分类间隔W的最大化也就是寻找$\boldsymbol{d}$的最大化。

看起来我们已经找到了目标函数的数学形式。但问题当然不会这么简单，我们还需要面对一连串令人头疼的麻烦。

3. 约束条件

虽然我们找到了目标函数，但是：

（1）我们如何判断一条直线能够将所有的样本点都正确分类？

（2）超平面的位置应该是在间隔区域的中轴线上，所以确定超平面位置的b参数也不能随意的取值。

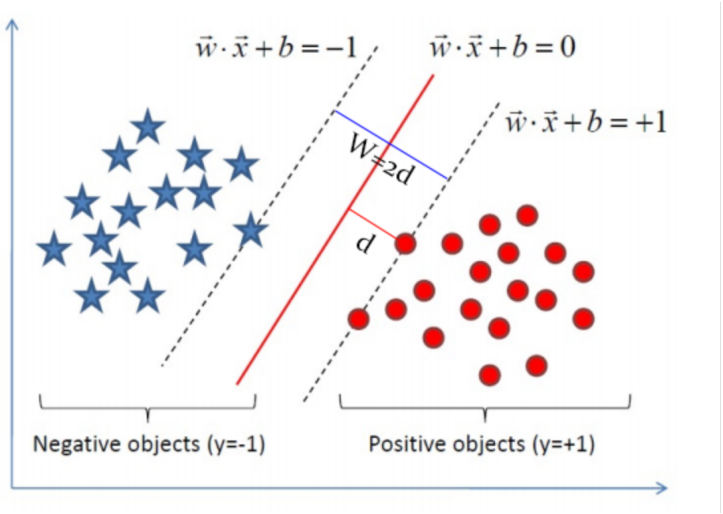
（3）对于一个给定的超平面，我们如何找到对应的支持向量，来计算距离d？

上述三个问题就是”约束条件“，也就是说，我们要优化的变量的取值范围收到了约束和限制。既然约束确实存在，那么就不得不用数学语言对它们进行描述。这里需要说明的是SVM可以通过一些小技巧，将这些约束条件糅合成一个不等式。请看下面糅合过程：

以下图为例，在平面空间中有红蓝两种点，对其分别标记为：

红色为正样本，标记为+1；

蓝色为负样本，标记为-1.



对每个样本点x_i加上类别标签y_i，则有

$$y_i = \begin{cases} +1 & \text{红} \\ -1 & \text{蓝} \end{cases}$$

如果我们的超平面能够完全将红蓝两种样本点分离开，那么则有

$$\begin{cases} \boldsymbol{\omega}^T \boldsymbol{x} + b > 0, y_i = 1 \\ \boldsymbol{\omega}^T \boldsymbol{x} + b < 0, y_i = -1 \end{cases} \quad (\text{公式}7)$$

如果要求在再高一点，假设超平面正好处于间隔区域的中轴线上，并且相应支持向量到超平面的距离为d,则公式可进一步写为：

$$\begin{cases} \frac{\boldsymbol{\omega}^T \boldsymbol{x}+b}{||\boldsymbol{\omega}||} \geq d, \ \forall\, y_i = +1 \\ \frac{\boldsymbol{\omega}^T \boldsymbol{x}+b}{||\boldsymbol{\omega}||} \leq -d, \ \forall\, y_i = -1 \end{cases} \quad (\text{公式}8)$$

符号∀是“对于所有满足条件的” 的缩写。也就是“任意一个”的意思。

对公式两边同时除以d，可得：

$$\begin{cases} \frac{\boldsymbol{\omega}_d^T \boldsymbol{x}+b_d}{||\boldsymbol{\omega}_d||} \geq 1, \ \forall\, y_i = +1 \\ \frac{\boldsymbol{\omega}_d^T \boldsymbol{x}+b_d}{||\boldsymbol{\omega}_d||} \leq -1, \ \forall\, y_i = -1 \end{cases} \quad (\text{公式}9)$$

其中，

$$\boldsymbol{\omega}_d = \frac{\boldsymbol{\omega}}{||\boldsymbol{\omega}||d}, \quad b_d = \frac{b}{||\boldsymbol{\omega}||d}$$

因为$||w||$和d都是标量。所以上述公式的两个矢量，依然描述一条直线的法向量和截距。所以下面两个公式，都是描述一条直线，数学模型代表的意义是一样的。

$$\boldsymbol{\omega}_d^T \boldsymbol{x} + b_d = 0$$
$$\boldsymbol{\omega}^T \boldsymbol{x} + b = 0$$

现在，让我们对w_d和b_d重新起个名字，就叫它们w和b，所以我们可得到：

$$\begin{cases} \boldsymbol{\omega}^T x_i + b \geq 1, \forall\, y_i = +1 \\ \boldsymbol{\omega}^T x_i + b \leq -1, \forall\, y_i = -1 \end{cases} \quad (\text{公式}10)$$

这个方程就是SVM最优化问题的约束条件。由于我们将标签定义为1和-1，所以此处我们可以将上述方程糅合成一个约束方程：

$$y_i(\boldsymbol{\omega}^T x_i + b) \geq 1, \forall x_i \quad (\text{公式}11)$$

4. 线性SVM优化问题基本描述

对于公式$\boldsymbol{\omega}^T x_i + b$ =1 or =-1，什么时候会发生呢？参考公式10 就会知道，只有当x_i是超平面的支持向量时，等于1或者-1的情况才会出现。无论是等于1还是-1，对于公式10来说，都有$|\boldsymbol{\omega}^T x_i + b| = 1$

所以对于这些支持向量来说：

$$d = \frac{|\boldsymbol{\omega}^T \boldsymbol{x}_i + b|}{||\boldsymbol{\omega}||} = \frac{1}{||\boldsymbol{\omega}||}, \forall \text{支持向量} x_i \quad (\text{公式}12)$$

我们原来的任务是找到一组参数$\boldsymbol{\omega}, b$使得分类间隔$W = 2d$最大化，根据公式12 就可以转变为$||\boldsymbol{\omega}||$的最小化问题，也等效于$\frac{1}{2}||\boldsymbol{\omega}||^2$的最小化问题。我们之所以要在$||\boldsymbol{\omega}||$上加上平方和1/2的系数，是为了以后进行最优化的过程中对目标函数求导时比较方便，但这绝不影响最优化问题最后的解。

所以，线性SVM最优化问题的数学描述就是：

$$\begin{aligned} &\min_{\boldsymbol{\omega},b} \ \frac{1}{2}||\boldsymbol{\omega}||^2 \\ &\text{s. t.} \ \ y_i(\boldsymbol{\omega}^T \boldsymbol{x}_i + b) \geq 1, \ \ i = 1, 2, \ldots, n \end{aligned} \quad (\text{公式}13)$$

这里n是样本点的总个数，缩写s. t. 表示“Subject to”，是“服从某某条件”的意思。公式13 描述的是一个典型的不等式约束条件下的二次型函数优化问题，同时也是支持向量机的基本数学模型。

5. 最优化问题的求解

通常我们需要求解的最优化问题有如下几类：

- 无约束优化问题，可以写为：

$$min\ f(x)$$

- 有等式约束的优化问题，可以写为：

$$\begin{aligned} &min\ f(x) \\ &s.t.\ h_i(x) = 0, \ \ i = 1, 2, \ldots, n \end{aligned}$$

- 有不等式约束的优化问题，可以写为：

$$
\begin{aligned}
&min\ f(x)\\
&s.t.\ g_i(x) \le 0,\ \ i = 1, 2, \dots, n\\
&h_j(x) = 0,\ \ j = 1, 2, \dots, m
\end{aligned}
$$

对于第(1)类的优化问题，尝试使用的方法就是**费马大定理**(Fermat)，即使用求取函数f(x)的导数，然后令其为零，可以求得候选最优值，再在这些候选值中验证；如果是凸函数，可以保证是最优解。这也就是我们高中经常使用的求函数的极值的方法。

对于第(2)类的优化问题，常常使用的方法就是**拉格朗日乘子法**（Lagrange Multiplier），即把等式约束$h_i(x)$用一个系数与f(x)写为一个式子，称为拉格朗日函数，而系数称为拉格朗日乘子。通过拉格朗日函数对各个变量求导，令其为零，可以求得候选值集合，然后验证求得最优值。

对于第(3)类的优化问题，常常使用的方法就是**KKT条件**(Karush-Kuhn-Tucker conditions)。同样地，我们把所有的等式、不等式约束与f(x)写为一个式子，也叫拉格朗日函数，系数也称拉格朗日乘子，通过一些条件，可以求出最优值的**必要条件**，这个条件称为KKT条件。对KTT条件感兴趣的可参考：

https://blog.csdn.net/james_616/article/details/72869015

必要条件和充要条件如果不理解，可以看下面这句话：

- A的**必要条件**就是A可以推出的**结论**
- A的**充分条件**就是可以推出A的**前提**

对于我们的线性SVM最优化问题：

$$
\begin{aligned}
&\min_{\omega,b}\ \frac{1}{2}||\boldsymbol{\omega}||^2 \\
&\text{s. t.}\ \ y_i(\boldsymbol{\omega}^T\boldsymbol{x}_i + b) \ge 1,\ \ i = 1, 2, \dots, n
\end{aligned}
\qquad (公式\,13)
$$

显然，它属于第(3)类的优化问题。那么在求解这类优化问题之前，我们还需要了解两个概念——拉格朗日函数和KKT条件。

6. 拉格朗日函数

首先，我们先要从宏观的视野上了解一下**拉格朗日对偶问题出现的原因和背景。**

我们知道我们要求解的是最小化问题，所以一个直观的想法是如果我能够构造一个函数，使得该函数在可行解区域内与原目标函数完全一致，而在可行解区域外的数值非常大，甚至是无穷大，那么这个**没有约束条件的新目标函数的优化问题**就与原来**有约束条件的原始目标函数的优化问题**是等价的问题。这就是使用拉格朗日方程的目的，它**将约束条件放到目标函数**中，从而将有约束优化问题转换为无约束优化问题。

但是对于拉格朗日函数，直接使用求导的方式求解仍然很困难，所以便有了**拉格朗日对偶**的诞生。

所以，显而易见的是，我们在拉格朗日优化我们的问题这个道路上，**需要进行下面二个步骤：**

- 将有约束的原始目标函数转换为无约束的新构造的拉格朗日目标函数
- 使用拉格朗日对偶性，将不易求解的优化问题转化为易求解的优化

第一步：**将有约束的原始目标函数转换为无约束的新构造的拉格朗日目标函数**

原始目标函数：

$$\min_{\boldsymbol{\omega},b}\ \frac{1}{2}||\boldsymbol{\omega}||^2 \qquad \text{（公式13）}$$
$$\text{s. t. }\ y_i(\boldsymbol{\omega}^T\boldsymbol{x}_i+b)\geq 1,\ \ i=1,2,\ldots,n$$

新构造的目标函数：

$$L(\boldsymbol{\omega},b,\boldsymbol{\alpha})=\frac{1}{2}||\boldsymbol{\omega}||^2-\sum_{i=1}^{n}\alpha_i(y_i(\boldsymbol{\omega}^T\boldsymbol{x}_i+b)-1)\quad \text{公式（14）}$$

其中α_i是拉格朗日乘子，且$\alpha_i\geq 0$，是我们人为设定的参数。

大家知道我们的目标是追求$\frac{1}{2}||\boldsymbol{\omega}||^2$的最小化，又因为

$$\alpha_i\geq 0$$
$$y_i(\boldsymbol{\omega}^T\boldsymbol{x}_i+b)\geq 1$$
$$y_i(\boldsymbol{\omega}^T\boldsymbol{x}_i+b)-1\geq 0$$
$$\sum_{i=1}^{n}\alpha_i(y_i(\boldsymbol{\omega}^T\boldsymbol{x}_i+b)-1)\geq 0$$

所以我们的新目标函数：

$$\min_{\boldsymbol{\omega},b}\left[\max_{\boldsymbol{\alpha}:\alpha_j\geq 0}L(\boldsymbol{\omega},b,\boldsymbol{\alpha})\right]\ \text{（公式15）}$$

第二步：**拉格朗日对偶函数**

对偶后的目标函数：

$$\max_{\boldsymbol{\alpha}:\alpha_j\geq 0}\left[\min_{\boldsymbol{\omega},b}L(\boldsymbol{\omega},b,\boldsymbol{\alpha})\right]\ \text{（公式16）}$$

接下来，我们就可以求解拉格朗日对偶函数了，求解出来的值就是我们最优化问题的结果，也就是可以得到最大间隔。

7. 对偶问题求解

第一步

根据公式16 ，我们可以先求$\min_{\boldsymbol{\omega},b}L(\boldsymbol{\omega},b,\boldsymbol{\alpha})$：

$$\min_{\boldsymbol{\omega},b}L(\boldsymbol{\omega},b,\boldsymbol{\alpha})=\min_{\boldsymbol{\omega},b}\left[\frac{1}{2}||\boldsymbol{\omega}||^2-\sum_{i=1}^{n}\alpha_i(y_i(\boldsymbol{\omega}^T\boldsymbol{x}_i+b)-1)\right]\quad \text{公式}(17)$$

分别令函数 $L(\boldsymbol{\omega},b,\boldsymbol{\alpha})$对 $\boldsymbol{\omega},b$求偏导，并使其等于0。

$$\frac{\partial L}{\partial \boldsymbol{\omega}}=0\implies \boldsymbol{\omega}=\sum_{i=1}^{n}\alpha_i y_i\boldsymbol{x}_i\ \text{（公式18）}$$

$$\frac{\partial L}{\partial b}=0\implies \sum_{i=1}^{n}\alpha_i y_i=0\ \text{（公式19）}$$

将公式18和公式19带入到公式17中：

$$
\begin{aligned}
L(\boldsymbol{\omega}, b, \boldsymbol{\alpha}) &= \frac{1}{2}||\boldsymbol{\omega}||^2 - \sum_{i=1}^{n} \alpha_i (y_i(\boldsymbol{\omega}^T \boldsymbol{x}_i + b) - 1) \\
&= \frac{1}{2}\boldsymbol{\omega}^T\boldsymbol{\omega} - \boldsymbol{\omega}^T \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i - b\sum_{i=1}^{n} \alpha_i y_i + \sum_{i=1}^{n} \alpha_i \\
&= \frac{1}{2}\boldsymbol{\omega}^T \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i - \boldsymbol{\omega}^T \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i - b * 0 + \sum_{i=1}^{n} \alpha_i \\
&= \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\boldsymbol{\omega}^T \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i \\
&= \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\left(\sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i\right)^T \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i \\
&= \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j \quad \text{公式}(20)
\end{aligned}
$$

从上面的最后一个式子，我们可以看出，此时的L(w,b,α)函数只含有一个变量，即α_i。

第二步

现在内侧的最小值求解完成，我们求解外侧的最大值，从上面的式子得到：

$$
\max_{\boldsymbol{\alpha}:\alpha_j \geq 0} \left[\min_{\boldsymbol{\omega}, b} L(\boldsymbol{\omega}, b, \boldsymbol{\alpha})\right]
$$

$$
\max_{\alpha} \left[\sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j\right]
$$

$$
s.t.\ \alpha_i \geq 0,\ i = 1, 2, \ldots, n
$$

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

现在我们的优化问题变成了如上的形式。至此，一切都很完美。但是这里有个假设：数据必须100%线性可分。但是，目前为止，我们知道几乎所有数据都不那么"干净"。这时我们就可以通过引入所谓的惩罚因子C，来允许有些数据点可以处于超平面的错误的一侧。此时，我们的目标函数不变，约束条件变为：

$$
s.t.\ C \geq \alpha_i \geq 0,\ i = 1, 2, \ldots, n
$$

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

惩罚因子C
C越大，表示分类越严格，允许错分的样本收到很大的限制越大，错分的样本数越少，越容易过拟合。

我们为什么要费这么大劲把优化问题转化成这样呢？实际上，是为了使用高效优化算法SMO算法。

三、SMO算法

1. 什么是SMO算法?

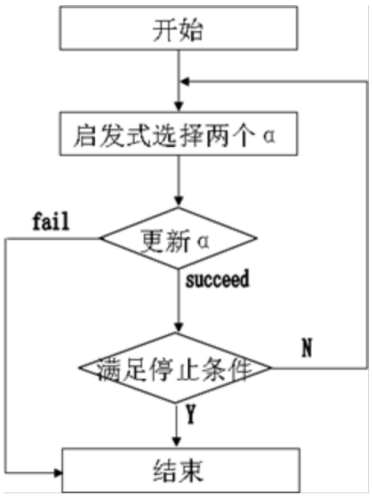
SMO算法就是序列最小优化（Sequential Minimal Optimization），它是由 John Platt 于1996年发布的专门用于训练SVM的一个强大算法。SMO算法的目的是将大优化问题分解为多个小优化问题来求解。这些小优化问题往往很容易求解，并且对它们进行顺序求解的结果与将它们作为整体来求解的结果完全一致的。在结果完全相同的同时，SMO算法的求解时间短很多。

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

SMO算法的工作原理是：每次循环中选择两个α进行优化处理。一旦找到了一对合适的α，那么就增大其中一个同时减小另一个。这里所谓的"合适"就是指两个α必须符合以下两个条件:

- 两个α必须要在间隔边界之外
- 这两个α还没有进行过区间化处理或者不在边界上。

2. SMO算法流程



关于SMO算法流程这一块内容，李航的《统计学习方法》中介绍的比较详细（$P_{124}-P_{133}$）,感兴趣的小伙伴可自行翻阅。这里我们就不再赘述一系列的推导过程，只给大家梳理一下SMO算法的实施步骤：

步骤1：计算误差：

$$E_i = f(x_i) - y_i = \left(\sum_{j=1}^{n} \alpha_j y_j \boldsymbol{x}_i^T \boldsymbol{x} + b\right) - y_i$$

步骤2：计算上下界H和L：

$$\begin{cases} L = max(0, \alpha_j^{old} - \alpha_i^{old}), H = min(C, C + \alpha_j^{old} - \alpha_i^{old}) \ \ if\ y_i \neq y_j \\ L = max(0, \alpha_j^{old} + \alpha_i^{old} - C), H = min(C, \alpha_j^{old} + \alpha_i^{old}) \ \ if\ y_i = y_j \end{cases}$$

步骤3：计算学习率η:

$$\eta = \boldsymbol{x}_i^T \boldsymbol{x}_i + \boldsymbol{x}_j^T \boldsymbol{x}_j - 2\boldsymbol{x}_i^T \boldsymbol{x}_j$$

步骤4：更新α_j:

$$\alpha_j^{new} = \alpha_j^{old} + \frac{y_i(E_i - E_j)}{\eta}$$

步骤5：根据取值范围修剪α_j:

$$\alpha^{new,clipped} = \begin{cases} H, & if\ \alpha_j^{new} \geq H \\ \alpha_j^{new}, & if\ L \leq \alpha_j^{new} \leq H \\ L, & if\ \alpha_j^{new} \leq L \end{cases}$$

步骤6：更新α_i：

$$\alpha_i^{new} = \alpha_i^{old} + y_i y_j (\alpha_j^{old} - \alpha_j^{new,clipped})$$

步骤7：更新b1和b2:

$$b_1^{new} = b^{old} - E_i - y_i(\alpha_i^{new} - \alpha_i^{old})\boldsymbol{x}_i^T\boldsymbol{x}_i - y_j(\alpha_j^{new} - \alpha_j^{old})\boldsymbol{x}_j^T\boldsymbol{x}_i$$
$$b_2^{new} = b^{old} - E_j - y_i(\alpha_i^{new} - \alpha_i^{old})\boldsymbol{x}_i^T\boldsymbol{x}_j - y_j(\alpha_j^{new} - \alpha_j^{old})\boldsymbol{x}_j^T\boldsymbol{x}_j$$

步骤8：根据b1和b2更新b:

$$b = \begin{cases} b_1 & 0 < \alpha_1^{new} < C \\ b_2 & 0 < \alpha_2^{new} < C \\ \frac{1}{2}(b_1 + b_2) & otherwise \end{cases}$$

根据求解出来的α^* 可计算原始问题的最优解w^*:

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

选择α^*的一个分量α_i^*满足条件$0 < \alpha_i^* < C$, 计算原始问题的最优解b^*:

$$b^* = y_j - \sum_{i=1}^{n} y_i \alpha_i^* (x_i \cdot x_j)$$

求解出原始最优化问题的解w^*, b^*,就得到线性支持向量机，其**分离超平面**为:

$$w^* \cdot x + b^* = 0$$
$$<=> \sum_{i=1}^{n} \alpha_i^* y_i (x_i \cdot x) + b^* = 0$$

分类决策函数为

$$f(x) = sign(w^* \cdot x + b^*)$$
$$<=> f(x) = sign\left(\sum_{i=1}^{n} \alpha_i^* y_i (x_i \cdot x) + b^*\right)$$

这里需要说明的是线性支持向量机的解w^*是唯一的，但b^*不一定唯一。因为对于α^*任意的一个分量α_i^*满足条件$0 < \alpha_i^* < C$都可以求出b^* ,从理论上讲，原始问题对b的解可能不唯一，但是在实际应用中，往往只会出现算法叙述的情况。

3. 简化版SMO算法

SMO算法的完整版实现需要大量的代码。这里我们先讨论SMO算法的简化版，主要用来正确理解这个算法的工作流程。然后会对这个简化版的SMO算法进行优化，加快它的运行速度。

3.1 SMO算法的伪代码

SMO函数的伪代码：

```
创建一个α向量并初始化为0向量
当迭代次数 < 最大迭代次数时（外循环）：
    对数据集中每个数据向量（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，则退出内循环
    如果所有向量都没被优化，迭代次数+1，继续下一次循环
```

3.2 构建辅助函数

生成特征矩阵和标签矩阵

```
import numpy as np
import pandas as pd
"""
函数功能：创建特征向量和标签向量
参数说明：
    file：原始文件路径
返回：
    xMat：特征向量
    yMat：标签向量
"""
def loadDataSet(file):
    dataSet= pd.read_table(file,header = None)
    xMat=np.mat(dataSet.iloc[:,:-1].values)
    yMat=np.mat(dataSet.iloc[:,-1].values).T
    return xMat,yMat

#导入数据集
file='testSet.txt'
xMat,yMat = loadDataSet(file)
```

数据集可视化

```
import matplotlib.pyplot as plt
%matplotlib inline

def showDataSet(xMat, yMat):
    data_p = []                                 #正样本
    data_n = []                                 #负样本
    m = xMat.shape[0]                           #样本总数
    for i in range(m):
        if yMat[i] > 0:
            data_p.append(xMat[i])
        else:
```

```
            data_n.append(xMat[i])
    data_p_ = np.array(data_p)                  #转换为numpy矩阵
    data_n_ = np.array(data_n)                  #转换为numpy矩阵
    plt.scatter(data_p_.T[0], data_p_.T[1])     #正样本散点图
    plt.scatter(data_n_.T[0], data_n_.T[1])     #负样本散点图
    plt.show()


 #运行函数，查看数据分布
showDataSet(xMat, yMat)
```

随机选择alpha对：

```
import random
"""
函数功能：随机选择一个索引
参数说明：
    i:第一个alpha索引
    m:数据集总行数
返回：
    j:随机选择的不与i相等的值
"""
def selectJrand(i,m):
    j=i
    while (j==i):
        j=int(random.uniform(0,m))
    return j
```

α_j的修剪函数：

```
"""
函数功能：修剪alpha_j
"""
def clipAlpha(aj,H,L):
    if aj>H:
        aj=H
    if L>aj:
        aj=L
    return aj
```

3.3 简化版SMO算法

```
"""
函数功能：
参数说明：
    xMat:特征向量
    yMat:标签向量
    C:常数
    toler:容错率
    maxIter:最大迭代次数
```

```
返回:
    b、alpha
"""
def smoSimple(xMat,yMat,C,toler,maxIter):
    b=0                                         #初始化b参数
    m,n=xMat.shape                              #m为数据集的总行数，n为特征的数量
    alpha = np.mat(np.zeros((m,1)))             #初始化alpha参数，设为0
    iters =0                                    #初始化迭代次数
    while (iters<maxIter):
        alpha_ = 0                              #初始化alpha优化次数
        for i in range(m):
            #步骤1：计算误差Ei
            fXi=np.multiply(alpha,yMat).T*(xMat*xMat[i,:].T)+b
            Ei=fXi-yMat[i]
            #优化alpha，设定容错率
            if ((yMat[i]*Ei<-toler)and(alpha[i]<C)) or
((yMat[i]*Ei>toler)and(alpha[i]>0)):
                #随机选择一个与alpha_i成对优化的alpha_j
                j=selectJrand(i,m)
                #步骤1：计算误差Ej
                fXj=np.multiply(alpha,yMat).T*(xMat*xMat[j,:].T)+b
                Ej=fXj-yMat[j]
                #保存更新前的alpha_i和alpha_j
                alphaIold=alpha[i].copy()
                alphaJold=alpha[j].copy()
                #步骤2：计算上下界H和L
                if (yMat[i]!=yMat[j]):
                    L=max(0,alpha[j]-alpha[i])
                    H=min(C,C+alpha[j]-alpha[i])
                else:
                    L=max(0,alpha[j]+alpha[i]-C)
                    H=min(C,C+alpha[j]+alpha[i])
                if L==H:
                    #print('L==H')
                    continue
                #步骤3：计算学习率eta(eta是alpha_j的最优修改量)
                eta=2*xMat[i,:]*xMat[j,:].T-xMat[i,:]*xMat[i,:].T-xMat[j,:]*xMat[j,:].T
                if eta>=0:
                    #print('eta>=0')
                    continue
                #步骤4：更新alpha_j
                alpha[j]-= yMat[j]*(Ei-Ej)/eta
                #步骤5：修剪alpha_j
                alpha[j]=clipAlpha(alpha[j],H,L)
                if abs(alpha[j]-alphaJold)<0.00001:
                    #print('alpha_j 变化太小')
                    continue
                #步骤6：更新alpha_i
                alpha[i]+=yMat[j]*yMat[i]*(alphaJold-alpha[j])
                #步骤7：更新b_1和b_2
                b1=b-Ei-yMat[i]*(alpha[i]-alphaIold)*xMat[i,:]*xMat[i,:].T-yMat[j]*
(alpha[j]-alphaJold)*xMat[i,:]*xMat[j,:].T
```

```
                b2=b-Ej-yMat[i]*(alpha[i]-alphaIold)*xMat[i,:]*xMat[j,:].T-yMat[j]*
(alpha[j]-alphaJold)*xMat[j,:]*xMat[j,:].T
                #步骤8：根据b_1和b_2更新b
                if (0<alpha[i])and(C>alpha[i]): b=b1
                elif (0<alpha[j])and(C>alpha[j]): b=b2
                else: b=(b1+b2)/2
                #统计优化次数
                alpha_+=1
                #print(f'第{iters}次迭代 样本{i},alpha优化次数:{alpha_}')
        #更新迭代次数
        if alpha_==0: iters+=1
        else: iters=0
        #print(f'迭代次数为:{iters}')
    return b,alpha
```

查看代码运行时间及结果：

```
%time b,alpha=smoSimple(xMat,yMat,0.6,0.001,5)
```

3.4 支持向量的可视化

```
"""
函数功能：提取出支持向量，用于后面画图
"""
def get_sv(xMat,yMat,alpha):
    m=xMat.shape[0]
    sv_x=[]
    sv_y=[]
    for i in range(m):
        if alpha[i]>0:
            sv_x.append(xMat[i])
            sv_y.append(yMat[i])
    sv_x1 =np.array(sv_x).T
    sv_y1 = np.array(sv_y).T
    return sv_x1,sv_y1

#运行函数
sv_x1,sv_y1=get_sv(xMat,yMat,alpha)
```

支持向量机绘图函数：

```
def showPlot(xMat, yMat,alpha,b):
    data_p = []                                 #正样本
    data_n = []                                 #负样本
    m = xMat.shape[0]                           #样本总数
    for i in range(m):
        if yMat[i] > 0:
            data_p.append(xMat[i])
        else:
            data_n.append(xMat[i])
```

```
    data_p_ = np.array(data_p)                    #转换为numpy矩阵
    data_n_ = np.array(data_n)                    #转换为numpy矩阵
    #样本散点图
    plt.scatter(data_p_.T[0], data_p_.T[1])       #正样本散点图
    plt.scatter(data_n_.T[0], data_n_.T[1])       #负样本散点图
    #绘制支持向量
    sv_x,sv_y=get_sv(xMat,yMat,alpha)
    plt.scatter(sv_x[0], sv_x[1], s=150, c='none', alpha=0.7, linewidth=1.5,
edgecolor='red')
    #绘制超平面
    w = np.dot((np.tile(np.array(yMat).reshape(1, -1).T, (1, 2)) * np.array(xMat)).T,
np.array(alpha))
    a1, a2 = w
    x1 = max(xMat[:,0])[0,0]
    x2 = min(xMat[:,0])[0,0]
    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])
    plt.show()

#运行函数
```

4. 完整版SMO算法

大家通过上面的代码就可以看出，我们只用了100个点组成的小规模数据集就已经耗费了十几分钟的时间。可想而知，如果我们使用更大一些的数据集的话，这个时间成本有多大。所以我们上面写的简化版SMO算法是不能直接拿来使用的，我们需要对简化版SMO算法的代码进行优化，因此就有了我们的完整版SMO算法。在这两个版本中，实现alpha的更改和代数运算的环节是一样的，在优化过程中唯一不同的是，选择alpha的方式。完整版的SMO算法应用了一些能够提速的启发方法。

什么叫启发式选择？【可参考】：http://www.dataguru.cn/thread-501427-1-1.html

4.1 构建辅助函数

先建立一个数据结构来保存我们所有的重要值，这里用的是**类**，这里用类并不是为了面向对象的编程，仅仅是作为一个数据结构来使用。我们构建一个仅包含init方法的optStruct类。该方法可以实现其成员变量的填充，省掉了手动输入的麻烦。

```
"""
数据结构，维护所有需要操作的值
参数说明:
    xMat: 特征矩阵
    yMat: 标签矩阵
    C: 松弛变量
    toler: 容错率
"""
class optStruct:
    def __init__(self, xMat, yMat, C, toler):
```

```
        self.X = xMat                                   #特征矩阵
        self.Y = yMat                                   #数据标签
        self.C = C                                      #松弛变量
        self.tol = toler                                #容错率
        self.m = xMat.shape[0]                          #特征矩阵行数
        self.alpha = np.mat(np.zeros((self.m,1)))       #根据矩阵行数初始化alpha参数为0
        self.b = 0                                      #初始化b参数为0
        self.eCa = np.mat(np.zeros((self.m,2)))         #根据矩阵行数初始化误差缓存，第一列为
是否有效的标志位，第二列为实际误差E的值。
```

对于给定的alpha值，我们构建辅助函数calcEk()来计算E值并返回。

```
"""
函数功能：计算误差
参数说明：
    oS：数据结构
    k：标号为k的数据
返回：
    Ek：标号为k的数据误差
"""

def calcEk(oS, k):
    fXk = np.multiply(oS.alpha,oS.Y).T*(oS.X*oS.X[k,:].T) + oS.b
    Ek = fXk - oS.Y[k]
    return Ek
```

辅助函数selectJ()用于选择内循环的alpha值。这里用的是启发式选择方法。

1）先将计算好的误差Ei储存到缓存区

2）挑选出所有不为0的alpha（nonzero()返回的是非零E值所对应的alpha值）

3）遍历所有不为0的alpha，找到改变最大的索引（步长最大化）

4）如果没有不为零的alpha,则遍历所有数据集（引用简化版中的随机选择alpha_j的索引值）

```
"""
内循环启发方式
参数说明：
    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
返回：
    j，maxK：标号为j或maxK的数据的索引值
    Ej：标号为j的数据误差
"""
def selectJ(i, oS, Ei):
    maxK = -1; maxDeltaE = 0; Ej = 0                #初始化
    oS.eCa[i] = [1,Ei]                              #根据Ei更新误差缓存
    eca = np.nonzero(oS.eCa[:,0].A)[0]              #返回误差不为0的数据的索引值
    if (len(eca)) > 1:                              #有不为0的误差
        for k in eca:                               #遍历,找到最大的Ek
```

```
            if k == i: continue                 #不计算i,浪费时间
            Ek = calcEk(oS, k)                  #计算Ek
            deltaE = abs(Ei - Ek)               #计算|Ei-Ek|
            if (deltaE > maxDeltaE):            #找到maxDeltaE
                maxK = k
                maxDeltaE = deltaE
                Ej = Ek
        return maxK, Ej                         #返回maxK,Ej
    else:                                       #没有不为0的误差
        j = selectJrand(i, oS.m)                #随机选择alpha_j的索引值
        Ej = calcEk(oS, j)                      #计算Ej
    return j, Ej                                #j,Ej
```

辅助函数updateEk()是用来计算误差值并存入缓存中。在对alpha优化之后会用到这个值。

```
"""
函数功能：计算Ek,并更新误差缓存
参数说明:
    oS：数据结构
    k：标号为k的数据的索引值
返回:无
"""
def updateEk(oS, k):
    Ek = calcEk(oS, k)          #计算Ek
    oS.eCa[k] = [1,Ek]          #更新误差缓存
```

4.2 寻找决策边界的优化例程

上面几个辅助函数本身的作用并不大，但是当和优化过程及外循环组合在一起的时候，就能够组成强大的完整版SMO算法了。

```
"""
函数功能：寻找决策边界的优化例程
参数说明:
    i：标号为i的数据的索引值
    oS：数据结构
返回:
    1：有任意一对alpha值发生变化
    0：没有任意一对alpha值发生变化或变化太小
"""
def innerL(i, oS):
    #步骤1：计算误差Ei
    Ei = calcEk(oS, i)
    #优化alpha,设定一定的容错率。
    if ((oS.Y[i] * Ei < -oS.tol) and (oS.alpha[i] < oS.C)) or ((oS.Y[i] * Ei > oS.tol)
and (oS.alpha[i] > 0)):
        #使用内循环启发方式选择alpha_j,并计算Ej
        j,Ej = selectJ(i, oS, Ei)
        #保存更新前的alpha值，使用深拷贝
```

```
        alphaIold = oS.alpha[i].copy()
        alphaJold = oS.alpha[j].copy()
        #步骤2：计算上下界L和H
        if (oS.Y[i] != oS.Y[j]):
            L = max(0, oS.alpha[j] - oS.alpha[i])
            H = min(oS.C, oS.C + oS.alpha[j] - oS.alpha[i])
        else:
            L = max(0, oS.alpha[j] + oS.alpha[i] - oS.C)
            H = min(oS.C, oS.alpha[j] + oS.alpha[i])
        if L == H:
            #print("L==H")
            return 0
        #步骤3：计算学习率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.alpha[j] -= oS.Y[j] * (Ei - Ej)/eta
        #步骤5：修剪alpha_j
        oS.alpha[j] = clipAlpha(oS.alpha[j],H,L)
        #更新Ej至误差缓存
        updateEk(oS, j)
        if (abs(oS.alpha[j] - alphaJold) < 0.00001):
            #print("alpha_j变化太小")
            return 0
        #步骤6：更新alpha_i
        oS.alpha[i] += oS.Y[j]*oS.Y[i]*(alphaJold - oS.alpha[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.Y[i]*(oS.alpha[i]-alphaIold)*oS.X[i,:]*oS.X[i,:].T -
oS.Y[j]*(oS.alpha[j]-alphaJold)*oS.X[i,:]*oS.X[j,:].T
        b2 = oS.b - Ej- oS.Y[i]*(oS.alpha[i]-alphaIold)*oS.X[i,:]*oS.X[j,:].T -
oS.Y[j]*(oS.alpha[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.alpha[i]) and (oS.C > oS.alpha[i]):
            oS.b = b1
        elif (0 < oS.alpha[j]) and (oS.C > oS.alpha[j]):
            oS.b = b2
        else:
            oS.b = (b1 + b2)/2
        return 1         #如果有任意一对alpha发生改变，则返回1
    else:
        return 0         #如果没有alpha对发生改变，则返回0
```

4.3 构建完整版SMO算法

当所有的辅助函数构建好之后，我们就可以来构建完整版Platt SMO算法了。

```
"""
完整的线性SMO算法
参数说明:
    xMat: 数据矩阵
    yMat: 数据标签
    C: 松弛变量
    toler: 容错率
    maxIter: 最大迭代次数
返回:
    oS.b: SMO算法计算的b
    oS.alpha: SMO算法计算的alpha
"""
def smoP(xMat, yMat, C, toler, maxIter):
    oS = optStruct(xMat, yMat, C, toler)                    #初始化数据结构
    iters = 0                                               #初始化当前迭代次数
    entireSet = True; alpha_ = 0
    #遍历整个数据集都alpha也没有更新或者超过最大迭代次数,则退出循环
    while (iters < maxIter) and ((alpha_ > 0) or (entireSet)):
        alpha_ = 0
        if entireSet:                                       #遍历整个数据集
            for i in range(oS.m):
                alpha_ += innerL(i,oS)                      #使用优化的SMO算法
                #print(f"全样本遍历:第{iters}次迭代 样本:{i}, alpha优化次数:{alpha_}")
            iters += 1
        else:                                               #遍历不在边界0和C的alpha
            nonBoundIs = np.nonzero((oS.alpha.A > 0) * (oS.alpha.A < C))[0]
            for i in nonBoundIs:
                alpha_ += innerL(i,oS)
                #print(f"非边界遍历:第{iters}次迭代 样本:{i}, alpha优化次数:{alpha_}")
            iters += 1
        if entireSet:                                       #遍历一次后改为非边界遍历
            entireSet = False
        elif (alpha_ == 0):                                 #如果alpha没有更新,计算全样本遍历
            entireSet = True
        #print(f"迭代次数: {iters}")
    return oS.b,oS.alpha                                    #返回SMO算法计算的b和alphas
```

测试函数运行结果及时间:

```
%time b,alpha =smoP(xMat, yMat, 0.6, 0.001, 100)
```

```
showPlot(xMat, yMat,alpha,b)
```

从图中可以看，完整版的SMO算法得出的支持向量变多了，而且进行100次的迭代也只耗费了0.1秒，进步是相当大了。

4.4 计算模型准确率

求解出原始最优化问题的解w^*, b^*,就得到线性支持向量机，其分离超平面为:

$$w^* \cdot x + b^* = 0$$

其中，$\omega^* = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i$

分类决策函数为

$$f(x) = sign(w^* \cdot x + b^*)$$

```
"""
函数功能：计算w
参数说明：
    xMat：特征矩阵
    yMat：标签矩阵
    alpha：alpha值
返回：
    w：计算得到的w
"""
def calcWs(alpha,xMat,yMat):
    m,n = xMat.shape
    w = np.zeros((n,1))
    for i in range(m):
        w += np.multiply(alpha[i]*yMat[i],xMat[i,:].T)  #w的计算公式
    return w
```

计算模型预测的准确率

```
"""
函数功能：计算模型准确率
参数说明：
    xMat:特征矩阵
    yMat:标签矩阵
    w:权重
    b:截距
返回：
    acc:模型预测准确率
"""
def calcAcc(xMat,yMat,w,b):
    yhat=[]
    re=0
    m,n = xMat.shape
    for i in range(m):
        result=xMat[i]*np.mat(w)+b    #超平面计算公式
        if result<0:
            yhat.append(-1)
        else:
            yhat.append(1)
        if yhat[i]==yMat[i]:
            re +=1
    acc = re/m
    print(f'模型预测准确率为{acc}')
    return acc
```

运行函数，查看模型预测准确率：

```
w = calcWs(alpha,xMat,yMat)
acc = calcAcc(xMat,yMat,w,b)
acc
```

从运行结果可以看出，模型预测的准确率非常高，基本上接近100%。

四、核函数

现实生活中的分类问题可能很多都是非线性的，然而非线性问题往往不好求解，所以我们希望能够用解决线性问题的方法来解决非线性问题，常采用的方法是非线性变换，就是将非线性问题转换为线性问题，这样我们就好求解啦。这个过程通常分为两步：

步骤1：使用一个变换将原空间的数据映射到新空间

步骤2：在新空间中用线性分类学习方法从训练数据中学习分类模型。

核技巧（kernel trick）就属于这样的方法。

1. 核技巧

核技巧其实是一种方法，可以用来计算$\Phi(x)^T\Phi(x)$。

核技巧的应用范围很广，通常情况下，只要需要进行高维转换并且需要计算点积，那么我们都可以使用核技巧来简化运算。此外，核技巧非常简单，它就是用低维空间的内积来求解高维空间的内积，省去了先做变换再做内积的过程，是计算变得简洁。

2. 核函数

假设ϕ是一个从低维的输入空间χ（欧式空间的子集或者离散集合）到高维的希尔伯特空间 H 的映射，那么如果存在函数 $K(x,z)$,对于任意的 $x,z\in\chi$,都有：

$$K(x,z)=\phi(x)\cdot\phi(z)$$

则称 $K(x,z)$ 为核函数（kernel function），$\phi(x)$ 称为映射函数，式中$\phi(x)\cdot\phi(z)$ 为$\phi(x)$ 和$\phi(z)$ 的内积。

同时我们不难发现，交换x,z的位置，则有$K(x,z)=K(z,x)$，这就是核函数的**对称性**。

大家再观察一下这个核函数，你会发现，这个核函数其实是在低维空间中计算的，这就是核函数的价值：它虽然也是将特征进行从低维到高维的转换，但是它的计算却是在低维空间进行的，并且将实质上的分类效果（利用了内积）表现在了高维空间中，这样既避免了直接在高维空间中的复杂计算，还解决了线性不可分的问题。

3. 核函数的常用类型

常用的核函数主要有：线性核函数、多项式核函数、高斯核函数、Sigmoid核函数、复合核函数。

3.1 线性核函数

线性核函数（Linear Kernel）是最简单的核函数，它直接计算两个输入特征向量的内积。

$$K(x,z)=x\cdot z$$

- 优点：简单高效，结果易解释，总能生成一个最简洁的线性分割超平面
- 缺点：只适用线性可分的数据集

3.2 多项式核函数

多项式核函数（Polynomial Kernel）是通过多项式来作为特征映射函数：

$$K(x,z)=(\gamma x\cdot z+c)^n$$

- 优点：可以拟合出复杂的分割超平面。
- 缺点：参数太多（γ,c,n），选择起来比较困难；另外多项式的阶数不宜太高否则会给模型求解带来困难。

3.3 高斯核函数

高斯核函数（Gaussian Kernel），在SVM中也称为 径向基核函数（Radial Basis Function,RBF），SKlearn 中的 libsvm 默认的核函数就是它。表达式为：

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

- 优点：可以把特征映射到无限多维，并且没有多项式计算那么困难，参数也比较好选择。
- 缺点：不容易解释，计算速度比较慢，容易过拟合。

3.4 Sigmoid核函数

Sigmoid核函数（Sigmoid Kernel）也是非线性SVM中常用的核函数之一，表达式为：

$$K(x,z)=tanh(\gamma x\cdot z+c)$$

其中，参数γ,c需要自行定义，tanh为双曲正切函数。

五、非线性SVM

前面的线性可分的数据我们基本上已经学会它的处理方法了，那对于非线性可分的数据呢？比如说像下面这种

```
#导入数据集1，查看数据分布
xMat,yMat = loadDataSet('testSetRBF.txt')
showDataSet(xMat, yMat)

#导入数据集2，查看数据分布
xMat,yMat = loadDataSet('testSetRBF2.txt')
showDataSet(xMat, yMat)
```

从运行结果可以看出，这两个数据集均为线性不可分。那对于这样的线性不可分数据集，我们就可以使用上述所讲到的核函数啦。我们这里使用的是径向基核函数。

1. 构建核转换函数

```
"""
函数功能：通过核函数将数据转换更高维的空间
参数说明：
```

```
    X：特征矩阵
    A：单个数据的向量
    kTup：包含核函数信息的元组
返回：
    K：计算的核K
"""
def kernelTrans(X, A, kTup):
    m,n = X.shape
    K = np.mat(np.zeros((m,1)))
    if kTup[0] == 'lin':
        K = X * A.T                             #线性核函数,只进行内积。
    elif kTup[0] == 'rbf':                      #高斯核函数,根据高斯核函数公式进行计算
        for j in range(m):
            deltaRow = X[j,:] - A
            K[j] = deltaRow*deltaRow.T
        K = np.exp(K/(-1*kTup[1]**2))           #计算高斯核K
    else:
        raise NameError('核函数无法识别')
    return K                                    #返回计算的核K
```

```
"""
数据结构，维护所有需要操作的值
参数说明：
    xMat：特征矩阵
    yMat：标签矩阵
    C：惩罚因子
    toler：容错率
    kTup：包含核函数信息的元组,第一个参数存放核函数类别，第二个参数存放必要的核函数需要用到的参数
 """
class optStruct:
    def __init__(self, xMat, yMat, C, toler, kTup):
        self.X = xMat                                       #特征矩阵
        self.Y = yMat                                       #标签矩阵
        self.C = C                                          #惩罚因子
        self.tol = toler                                    #容错率
        self.m = xMat.shape[0]                              #特征矩阵行数
        self.alpha = np.mat(np.zeros((self.m,1)))           #根据矩阵行数初始化alpha参数为0
        self.b = 0                                          #初始化b参数为0
        self.eCa = np.mat(np.zeros((self.m,2)))             #根据矩阵行数初初始化误差缓存，第一列
为是否有效的标志位，第二列为实际的误差E的值。
        self.K = np.mat(np.zeros((self.m,self.m)))          #初始化核K
        for i in range(self.m):                             #计算所有数据的核K
            self.K[:,i] = kernelTrans(self.X, self.X[i,:], kTup)
```

2. 更新辅助函数

```
"""
函数功能：计算误差
参数说明：
    oS：数据结构
    k：标号为k的数据
返回：
    Ek：标号为k的数据误差
"""
def calcEk(oS, k):
    fXk = np.multiply(oS.alpha,oS.Y).T*oS.K[:,k] + oS.b
    Ek = fXk - oS.Y[k]
    return Ek
```

在更新函数innerL()时，其实我们只更新了eta、b1、b2三个计算公式

```
"""
更新的寻找决策边界的优化例程
参数说明：
    i：标号为i的数据的索引值
    oS：数据结构
返回：
    1：有任意一对alpha值发生变化
    0：没有任意一对alpha值发生变化或变化太小
"""
def innerL(i, oS):
    #步骤1：计算误差Ei
    Ei = calcEk(oS, i)
    #优化alpha,设定一定的容错率。
    if ((oS.Y[i] * Ei < -oS.tol) and (oS.alpha[i] < oS.C)) or ((oS.Y[i] * Ei > oS.tol)
and (oS.alpha[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aplpha值，使用深拷贝
        alphaIold = oS.alpha[i].copy()
        alphaJold = oS.alpha[j].copy()
        #步骤2：计算上下界L和H
        if (oS.Y[i] != oS.Y[j]):
            L = max(0, oS.alpha[j] - oS.alpha[i])
            H = min(oS.C, oS.C + oS.alpha[j] - oS.alpha[i])
        else:
            L = max(0, oS.alpha[j] + oS.alpha[i] - oS.C)
            H = min(oS.C, oS.alpha[j] + oS.alpha[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.alpha[j] -= oS.Y[j] * (Ei - Ej)/eta
```

```
        #步骤5：修剪alpha_j
        oS.alpha[j] = clipAlpha(oS.alpha[j],H,L)
        #更新Ej至误差缓存
        updateEk(oS, j)
        if (abs(oS.alpha[j] - alphaJold) < 0.00001):
            #print("alpha_j变化太小")
            return 0
        #步骤6：更新alpha_i
        oS.alpha[i] += oS.Y[j]*oS.Y[i]*(alphaJold - oS.alpha[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.Y[i]*(oS.alpha[i]-alphaIold)*oS.K[i,i]- oS.Y[j]*
(oS.alpha[j]-alphaJold)*oS.K[i,j]
        b2 = oS.b - Ej- oS.Y[i]*(oS.alpha[i]-alphaIold)*oS.K[i,j]- oS.Y[j]*
(oS.alpha[j]-alphaJold)*oS.K[j,j]
        #步骤8：根据b_1和b_2更新b
        if (0 < oS.alpha[i]) and (oS.C > oS.alpha[i]):
            oS.b = b1
        elif (0 < oS.alpha[j]) and (oS.C > oS.alpha[j]):
            oS.b = b2
        else:
            oS.b = (b1 + b2)/2
        return 1
    else:
        return 0
```

3. 非线性SVM算法

更新完整版SMO算法

```
"""
更新的完整版线性SMO算法（加入线性核函数）
参数说明：
    xMat：数据矩阵
    yMat：数据标签
    C：松弛变量
    toler：容错率
    maxIter：最大迭代次数
    kTup：包含核函数信息的元组
返回：
    oS.b：SMO算法计算的b
    oS.alpha：SMO算法计算的alpha
"""
def smoP(xMat, yMat, C, toler, maxIter,kTup = ('lin',0)):
    oS = optStruct(xMat, yMat, C, toler,kTup)                #初始化数据结构
    iters = 0                                                #初始化当前迭代次数
    entireSet = True; alpha_ = 0
    #遍历整个数据集都alpha也没有更新或者超过最大迭代次数,则退出循环
    while (iters < maxIter) and ((alpha_ > 0) or (entireSet)):
        alpha_ = 0
```

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

            for i in range(oS.m):
                alpha_ += innerL(i,oS)                  #使用优化的SMO算法
                #print(f"全样本遍历:第{iters}次迭代 样本:{i}, alpha优化次数:{alpha_}")
            iters += 1
        else:                                           #遍历不在边界0和C的alpha
            nonBoundIs = np.nonzero((oS.alpha.A > 0) * (oS.alpha.A < C))[0]
            for i in nonBoundIs:
                alpha_ += innerL(i,oS)
                #print(f"非边界遍历:第{iters}次迭代 样本:{i}, alpha优化次数:{alpha_}")
            iters += 1
        if entireSet:                                   #遍历一次后改为非边界遍历
            entireSet = False
        elif (alpha_ == 0):                             #如果alpha没有更新,计算全样本遍历
            entireSet = True
        #print(f"迭代次数: {iters}")
    return oS.b,oS.alpha                                #返回SMO算法计算的b和alphas
```

测试函数运行结果：

```
%time b,alpha =smoP(xMat, yMat, 0.6, 0.001, 40,kTup = ('lin',0))
```

4. 利用核函数进行分类

```
"""
利用核函数进行分类的径向基测试函数
参数说明:
    k1：使用高斯核函数的时候表示到达率
返回:无
"""
def testRbf(k1 = 1.3):
    xMat,yMat = loadDataSet('testSetRBF.txt')                   #加载训练集
    b,alpha = smoP(xMat, yMat, 200, 0.0001, 100, ('rbf', k1))  #根据训练集计算b和alpha
    svInd = np.nonzero(alpha.A > 0)[0]                          #获得支持向量的索引
    sVs = xMat[svInd]                                           #获得支持向量
    labelSV = yMat[svInd]                                       #获得支持向量的标签
    print(f"支持向量个数:{sVs.shape[0]}")
    m,n = xMat.shape
    errorCount = 0
    for i in range(m):
        K = kernelTrans(sVs,xMat[i,:],('rbf', k1))              #计算各个点的核
        predict = K.T * np.multiply(labelSV,alpha[svInd]) + b   #根据支持向量的点，预测结
果
        if np.sign(predict) != np.sign(yMat[i]):                #sign()函数功能是x<0则
返回-1，x=0返回0，x>0返回1
            errorCount += 1                                     #预测错误的个数
    acc_train = 1-errorCount/m                                  #计算准确率
    #print(f"训练集准确率为：{acc_train}")
    xMat,yMat = loadDataSet('testSetRBF2.txt')                  #加载测试集
```

```
    errorCount = 0
    m,n = xMat.shape
    for i in range(m):
        K = kernelTrans(sVs,xMat[i,:],('rbf', k1))                #此处支持向量sVs是根据训
练集得出的
        predict=K.T * np.multiply(labelSV,alpha[svInd]) + b       #此处的支持向量的标签
labelSV、alpha、b都是根据训练集计算出的
        if np.sign(predict) != np.sign(yMat[i]):
            errorCount += 1
    acc_test = 1-errorCount/m
    #print(f"测试集准确率为：{acc_test}")
    return acc_train,acc_test
```

保持k1=1.3的情况下，绘制运行10次的结果：

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['font.sans-serif']=['simhei']

train_acc=[]
test_acc=[]
for i in range(10):
    a,b = testRbf(k1 = 1.3)
    train_acc.append(a)
    test_acc.append(b)
plt.plot(range(1,11),train_acc)
plt.plot(range(1,11),test_acc)
plt.xlabel('次数')
plt.ylabel('模型准确率')
plt.legend(['train_acc','test_acc'])
plt.show()
```

结果可以看出，支持向量的个数基本维持在26上下波动，模型在训练集上的预测准确率在90%-100%，而在测试集上的表现略低于在训练集上的表现，在测试集上的准确率基本在82%-95%

将k1从0.1调到1.3，查看运行结果

```
train_acc=[]
test_acc=[]
for k1 in np.arange(0.1,1.4,0.1):
    a,b=testRbf(k1)
    train_acc.append(a)
    test_acc.append(b)
plt.plot(np.arange(0.1,1.4,0.1),train_acc)
plt.plot(np.arange(0.1,1.4,0.1),test_acc)
plt.xlabel('k1')
plt.ylabel('accuracy')
plt.legend(['train_acc','test_acc'])
plt.show()
```

六、SVM之手写数字识别

此处我们使用SVM进行二分类，所以对digit数据集进行了简单处理，我们只保留了数字1和9，并且将数字9定义为负样本，标签为-1，将数字1定义为正样本，标签为1.

1. 导入数据集

这里的代码与第1期k-近邻算法里面的代码稍有不同，根据需要进行了一定的修改。这里我们将文件导进来之后不再形成一个dataframe形式的数据集，而是形成了两个矩阵，一个特征矩阵，一个标签矩阵，为了匹配上面所写的SMO算法。

获取特征矩阵和标签矩阵

```
import os

"""
函数功能：得到特征矩阵和标签矩阵
参数说明：
    path:文件夹的路径
"""
def get_Mat(path):
    FileList = os.listdir(path)                 #提取出文件夹中所有文件的名字
    m = len(FileList)                           #文件个数
    label = []                                  #初始化分类标签
    xMat = np.mat(np.zeros((m,1024)))           #初始化一个m*1024的全零矩阵
    for i in range(m):                          #遍历每一个文件
        xMat_i = np.mat(np.zeros((1,1024)) )    #初始化一个1*1024的全零矩阵
        filename = FileList[i]                  #提取出当前文件名
        #读取文件，结果是一个32行*每一行中是一个32位字符串的dataframe
        txt = pd.read_csv(f'{path}/{filename}',header=None)
        for j in range(32):                     #遍历当前文件的每一行
            num = txt.iloc[j,:]                 #当前行中的字符串
            for k in range(32):                 #遍历字符串中的每一个数据
                xMat_i[0,32*j+k]=int(num[0][k])  #将1*1024矩阵中相应位置替换为当前字符串的
值，注意要转换成int类型
        xMat[i,:] = xMat_i                      #将m*1024矩阵中相应的位置替换为该文件的矩阵
        filelabel = int(filename.split('_')[0]) #根据文件名切分出手写数字的值
        if filelabel==9:                        #将数字9的标签定为-1
            label.append(-1)
        else:                                   #将数字1的标签定为1
            label.append(1)
    yMat=np.mat(label).T                        #将标签列转换为矩阵
    return xMat,yMat
```

查看运行结果，注意一定要确定得到的是两个矩阵

```
path='digits/trainingDigits'
xMat,yMat=get_Mat(path)
xMat
xMat.shape

yMat[:5]
```

2. 手写数字的测试函数

```
"""
手写数字的测试函数
参数说明:
    kTup: 包含核函数信息的元组
返回:
    acc_train: 训练集的准确率
    acc_test: 测试集的准确率
    sVs.shape[0]: 支持向量的个数
"""
def testDigits(kTup=('rbf', 10)):
    xMat,yMat = get_Mat('digits/trainingDigits')                    #得到训练集的特征矩阵和
标签矩阵
    b,alpha = smoP(xMat, yMat, 200, 0.0001, 10000, kTup)            #根据训练集计算出b和
alpha
    svInd = np.nonzero(alpha.A>0)[0]                                #提取出非零alpha的索引
(也就是支持向量的索引)
    sVs=xMat[svInd]                                                 #提取出支持向量的特征矩
阵
    labelSV = yMat[svInd]                                           #提取出支持向量的标签矩
阵
    print(f"支持向量个数:{sVs.shape[0]}" )
    m,n = xMat.shape                                                #m是数据集总行数，n是特
征数量
    errorCount = 0                                                  #初始化错误数
    for i in range(m):                                              #遍历每一条数据
        K = kernelTrans(sVs,xMat[i,:],kTup)                         #进行数据转换
        predict=K.T * np.multiply(labelSV,alpha[svInd]) + b         #根据支持向量计算预测结
果
        if np.sign(predict) != np.sign(yMat[i]):                    #检查预测结果与原始标签
是否一样
            errorCount += 1                                         #统计预测错误的个数
    acc_train=1-errorCount/m                                        #计算训练集的准确率
    #print(f"训练集准确率: {acc_train}")
    xMat,yMat = get_Mat('digits/testDigits')
    errorCount = 0
    m,n = xMat.shape
    for i in range(m):
        K = kernelTrans(sVs,xMat[i,:],kTup)
        predict=K.T * np.multiply(labelSV,alpha[svInd]) + b
        if np.sign(predict) != np.sign(yMat[i]):
            errorCount += 1
    acc_test=1-errorCount/m
    #print(f"训练集准确率: {acc_test}")
```

```
    return acc_train,acc_test,svs.shape[0]
```

3. 不同核函数及参数运行结果

我们尝试改变核函数的类型以及参数，来看看结果有何不同

```
acc_train=[]
acc_test=[]
SVnum=[]
kTups = [('rbf',0.1),('rbf',5),('rbf',10),('rbf',50),('rbf',100),('lin',0)]
for kTup in kTups:
    a,b,c = testDigits(kTup)
    acc_train.append(a)
    acc_test.append(b)
    SVnum.append(c)
```

将结果变成一个DF

```
df = pd.DataFrame({'内核设置':kTups,
                   '训练准确率':acc_train,
                   '测试准确率':acc_test,
                   '支持向量数':SVnum})

#将结果进行简单处理，使数据更好看一些
for i in df.columns[1:-1]:
df.loc[:,i]=[round(x*100,1) for x in df.loc[:,i].values ]

#查看结果
df
```

从结果可以看出，参数取10的时候模型在训练集和测试集上的表现都不错。

这里有两个比较有趣的地方是：

1. 线性核函数无论是在训练集还是测试集上的表现也不差，并且支持向量数相对来说更少一些
2. 径向基核函数在训练集上的表现随着支持向量数的增加而变得越来越好

七、算法总结

支持向量机是一个非常强大的分类方法。在集成学习和神经网络之类的算法没有表现出优越性能前，SVM基本占据了分类模型的统治地位。目前在大数据背景下，SVM由于其在大样本时的超级计算量，热度有所下降，但仍然是一个常用的机器学习算法。

1. 优点

SVM的主要优点有：

- 模型依赖的支持向量比较少，无需依赖全部数据
- 解决高维特征的分类问题和回归问题很有效，在特征维度大于样本数时依然有很好的效果
- 有大量的核函数可以使用，从而可以很灵活的处理各种非线性的分类回归问题
- 一旦模型训练完成，预测阶段的速度非常快

2. 缺点

SVM的主要缺点有：

- 大样本学习的计算成本会非常高
- 模型的训练效果非常依赖于惩罚因子C的选择
- 非线性问题的核函数的选择没有通用标准，难以选择一个合适的核函数
- SVM对缺失值比较敏感

其他

- 菊安酱的直播间：https://live.bilibili.com/14988341
- 下周一（2018/12/10）将讲解**Adaboost算法**，欢迎各位进入菊安酱的直播间观看直播
- 如有问题，可以给我留言哦~

参考资料：

[1] 5岁小孩也能看懂的SVM：

http://bytesizebio.net/2014/02/05/support-vector-machines-explained-well/

[2] 大侠与魔鬼的故事：

https://www.zhihu.com/question/21094489/answer/86273196

[3] 陈老师的知乎专栏：

https://zhuanlan.zhihu.com/p/24638007

[4] 深入理解拉格朗日乘子法和KKT条件

https://blog.csdn.net/xianlingmao/article/details/7919597

[5] Jack大佬的博客：

https://cuijiahua.com/blog/2017/11/ml_8_svm_1.html

[6]SVM支持向量机中Platt-SMO解法"启发式选择变量"的解释

http://www.dataguru.cn/thread-501427-1-1.html