《机器学习实战》学习笔记（六）：支持向量机

原创 我是管小亮 2019-08-18 16:22:12 2755 收藏 10 版权

分类专栏： Machine Learning♡ 文章标签： 机器学习 机器学习实战 读书笔记及代码 支持向量机

欢迎关注WX公众号：【程序员管小亮】

【机器学习】《机器学习实战》读书笔记及代码 总目录

- https://blog.csdn.net/TeFuirnever/article/details/99701256

GitHub代码地址：

- https://github.com/TeFuirnever/Machine-Learning-in-Action

目录

本章内容

- 简单介绍支持向量机
- 利用SMO进行优化
- 利用核函数对数据进行空间转换
- 将SVM和其他分类器进行对比

“由于理解支持向量机（Support Vector Machines，SVM）需要掌握一些理论知识，而这对于读者来说有一定难度，于是建议读者直接下载LIBSVM使用。”这句话我在很多书上看到相关介绍，包括机器学习实战中的理论也不是很多，西瓜书相对多一些但是不够详细，所以想要仔细看理论的话，可以移步：

- 【机器学习】《机器学习》周志华西瓜书读书笔记：第6章 - 支持向量机
- 【机器学习】《机器学习》周志华西瓜书习题参考答案：第6章 - 支持向量机

关于现在还有必要对SVM深入学习吗？这个引用一个知乎高赞回答。

> 窃以为，学东西，特别是科学知识不应以“火不火”作为评价标准。而是应以点带面、举一反三的建立学术知识体系，简单单纯来说也就是积累。不要看现在SVM被DL盖过了风头就产生“学了会没用，会吃亏”的念头。实际上，SVM整套体系，不论是其理论基础还是具体实现（如经典的libsvm和liblinear），都有其研究的价值。万变不离其宗，万事万物的“道”是相通的，说不准以后会在解决某些问题的时候借鉴到曾经研究过的SVM相关思想或实现技巧，彼时，便会产生融会贯通之畅快感。

闲言少叙，开始正题，本文主要关注 **序列最小优化（Sequential Minimal Optimization，SMO）** 算法，一种求解支持向量机二次规划的算法。

1、基于最大间隔分隔数据

> 支持向量机
>
> - 优点：泛化错误率低，计算开销不大，结果易解释。
> - 缺点：对参数调节和核函数的选择敏感，原始分类器不加修改仅适用于处理二类问题。
> - 适用数据类型：数值型和标称型数据。

在介绍SVM这个主题之前，先解释几个概念。

- **线性可分**：可以很容易就在数据中给出一条直线将两组数据点分开。

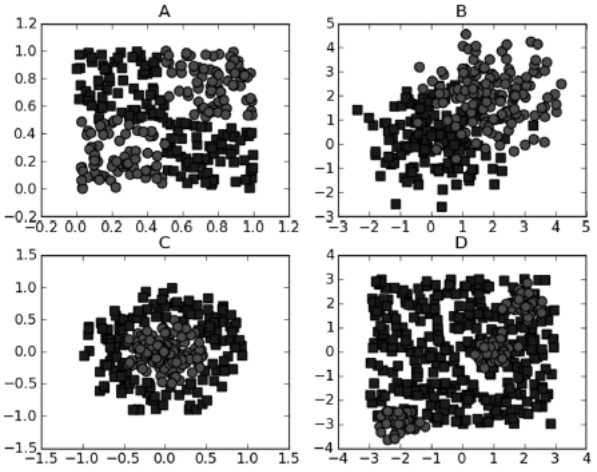


图6-1　4个线性不可分的数据集

上图的数据都是混合在一起，也就是不能用一条直线进行分类的数据，所以也就是线性不可分数据。

- **分隔超平面**：将数据集分割开来的直线。

在上面的例子中，由于数据点都在二维平面上，所以此时分隔超平面就只是一条直线。但是，如果所给的数据集是三维的，那么此时用来分隔数据的就是一个平面。显而易见，更高维的情况可以依此类推。如果数据集是1024维的，那么就需要一个1023维的某某对象来对数据进行分隔。这个1023维的某某对象到底应该叫什么？N-1维呢？该对象被称为超平面（hyperplane），也就是分类的决策边界。

理想状态是分布在超平面一侧的所有数据都属于某个类别，而分布在另一侧的所有数据则属于另一个类别。

- **间隔**：离分隔超平面最近的点，到分隔面的距离。

间隔应该尽可能地大，这是因为如果我们犯错或者在有限数据上训练分类器的话，大的间隔可以增加分类器的鲁棒性。

- **支持向量**：离分隔超平面最近的那些点。

支持向量到分割面的距离应该最大化。

2、寻找最大间隔

如何求解数据集的最佳分隔直线?

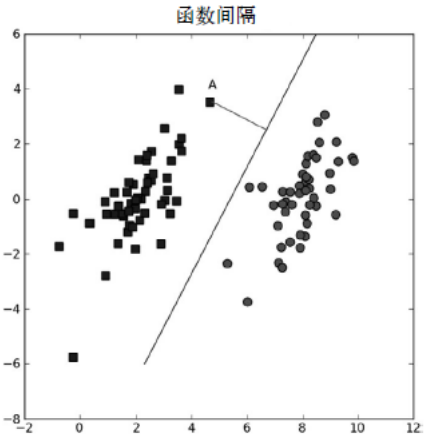


图6-3　点A到分隔平面的距离就是该点到分隔面的法线长度

推导公式为:

$$|\mathbf{w}^{\mathrm{T}}\mathbf{A}+b|/||\mathbf{w}||$$

最大化间隔的目标就是找出分类器定义中的w和b。为此，我们必须找到具有最小间隔的数据点，而这些数据点也就是前面提到的支持向量。一旦找到具有最小间隔的数据点，我们就需要对该间隔最大化。这就可以写作:

$$\arg\max_{w,b}\left\{\min_{n}(\text{label}\cdot(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+b))\cdot\frac{1}{\|\boldsymbol{w}\|}\right\}$$

直接求解上述问题相当困难，所以我们将它转换成为另一种更容易求解的形式。

$$\max_{\alpha}\left[\sum_{i=1}^{m}\alpha-\frac{1}{2}\sum_{i,j=1}^{m}\text{label}^{(i)}\cdot\text{label}^{(j)}\cdot a_i\cdot a_j\left\langle x^{(i)},x^{(j)}\right\rangle\right]$$

约束条件为:

$$C\geqslant\alpha\geqslant 0\text{，和}\sum_{i-1}^{m}\alpha_i\cdot\text{label}^{(i)}=0$$

其中常数C 用于控制 “最大化间隔” 和 “保证大部分点的函数间隔小于1.0” 这两个目标的权重。在优化算法的实现代码中，常数C 是一个参数，因此可以通过调节该参数得到不同的结果。一旦求出了所有的alpha，那么分隔超平面就可以通过这些alpha 来表达。

到目前为止，已经了解了一些理论知识，但是比较理论总归要回到实践上，这也是每个算法的归宿所在，通过编程，在数据集上将这些理论付诸实践。

> SVM的一般流程
> (1) 收集数据：可以使用任意方法。
> (2) 准备数据：需要数值型数据。
> (3) 分析数据：有助于可视化分隔超平面。
> (4) 训练算法：SVM的大部分时间都源自训练，该过程主要实现两个参数的调优。
> (5) 测试算法：十分简单的计算过程就可以实现。
> (6) 使用算法：几乎所有分类问题都可以使用SVM，值得一提的是，SVM本身是一个二类分类器，对多类问题应用SVM需要对代码做一些修改。

3、简化版 SMO 算法

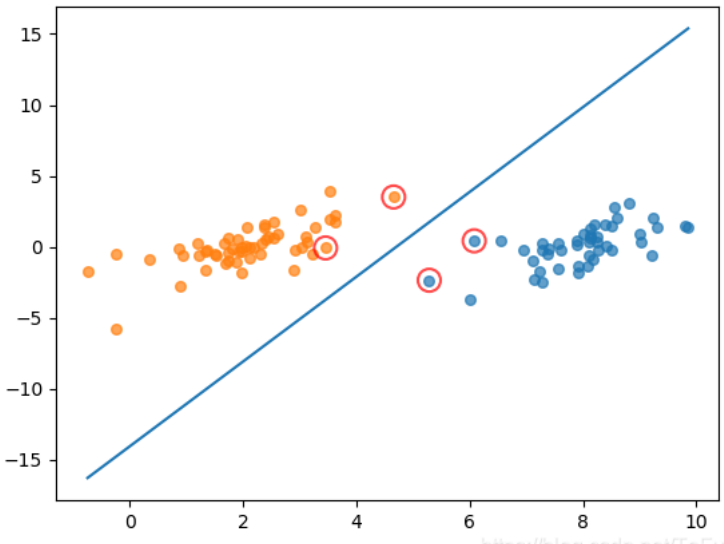
简化版SMO算法，省略了确定要优化的最佳alpha 对的步骤，而是首先在数据集上进行遍历每一个alpha，再在剩余的数据集中找到另外一个alpha，构成要优化的alpha 对，同时对其进行优化，这里的同时是要确保公式：$\Sigma\alpha_i * label^{(i)}=0$。

所以改变一个alpha 显然会导致等式失效，所以这里需要同时改变两个alpha。接下来看实际的代码:

```python
# -*- coding:UTF-8 -*-
from time import sleep
import matplotlib.pyplot as plt
import numpy as np
import random
import types

"""
函数说明:读取数据

Parameters:
    fileName - 文件名
Returns:
    dataMat - 数据矩阵
    labelMat - 数据标签
"""
def loadDataSet(fileName):
    dataMat = []; labelMat = []
    fr = open(fileName)
    for line in fr.readlines():                                    #逐行读取，滤除空格等
        lineArr = line.strip().split('\t')
        dataMat.append([float(lineArr[0]), float(lineArr[1])])     #添加数据
        labelMat.append(float(lineArr[2]))                         #添加标签
    return dataMat,labelMat


"""
函数说明:随机选择alpha

Parameters:
    i - alpha
    m - alpha参数个数
Returns:
    j -
"""
def selectJrand(i, m):
    j = i                                 #选择一个不等于i的j
    while (j == i):
        j = int(random.uniform(0, m))
    return j

"""
函数说明:修剪alpha

Parameters:
    aj - alpha值
    H - alpha上限
    L - alpha下限
Returns:
    aj - alpah值
"""
def clipAlpha(aj,H,L):
    if aj > H:
        aj = H
    if L > aj:
        aj = L
    return aj

"""
函数说明:简化版SMO算法

Parameters:
    dataMatIn - 数据矩阵
    classLabels - 数据标签
    C - 松弛变量
    toler - 容错率
    maxIter - 最大迭代次数
Returns:
    无
"""
def smoSimple(dataMatIn, classLabels, C, toler, maxIter):
    #转换为numpy的mat存储
    dataMatrix = np.mat(dataMatIn); labelMat = np.mat(classLabels).transpose()
    #初始化b参数，统计dataMatrix的维度
    b = 0; m,n = np.shape(dataMatrix)
    #初始化alpha参数，设为0
    alphas = np.mat(np.zeros((m,1)))
    #初始化迭代次数
```

```
    iter_num = 0
    #最多迭代matIter次
    while (iter_num < maxIter):
        alphaPairsChanged = 0
        for i in range(m):
            #步骤1：计算误差Ei
            fXi = float(np.multiply(alphas,labelMat).T*(dataMatrix*dataMatrix[i,:].T)) + b
            Ei = fXi - float(labelMat[i])
            #优化alpha，更设定一定的容错率。
            if ((labelMat[i]*Ei < -toler) and (alphas[i] < C)) or ((labelMat[i]*Ei > toler) and (alphas[i] > 0)):
                #随机选择另一个与alpha_i成对优化的alpha_j
                j = selectJrand(i,m)
                #步骤1：计算误差Ej
                fXj = float(np.multiply(alphas,labelMat).T*(dataMatrix*dataMatrix[j,:].T)) + b
                Ej = fXj - float(labelMat[j])
                #保存更新前的aplpha值，使用深拷贝
                alphaIold = alphas[i].copy(); alphaJold = alphas[j].copy();
                #步骤2：计算上下界L和H
                if (labelMat[i] != labelMat[j]):
                    L = max(0, alphas[j] - alphas[i])
                    H = min(C, C + alphas[j] - alphas[i])
                else:
                    L = max(0, alphas[j] + alphas[i] - C)
                    H = min(C, alphas[j] + alphas[i])
                if L==H: print("L==H"); continue
                #步骤3：计算eta
                eta = 2.0 * dataMatrix[i,:]*dataMatrix[j,:].T - dataMatrix[i,:]*dataMatrix[i,:].T - dataMatrix[j,:]*dataMatrix[j
                if eta >= 0: print("eta>=0"); continue
                #步骤4：更新alpha_j
                alphas[j] -= labelMat[j]*(Ei - Ej)/eta
                #步骤5：修剪alpha_j
                alphas[j] = clipAlpha(alphas[j],H,L)
                if (abs(alphas[j] - alphaJold) < 0.00001): print("alpha_j变化太小"); continue
                #步骤6：更新alpha_i
                alphas[i] += labelMat[j]*labelMat[i]*(alphaJold - alphas[j])
                #步骤7：更新b_1和b_2
                b1 = b - Ei- labelMat[i]*(alphas[i]-alphaIold)*dataMatrix[i,:]*dataMatrix[i,:].T - labelMat[j]*(alphas[j]-alphaJ
                b2 = b - Ej- labelMat[i]*(alphas[i]-alphaIold)*dataMatrix[i,:]*dataMatrix[j,:].T - labelMat[j]*(alphas[j]-alphaJ
                #步骤8：根据b_1和b_2更新b
                if (0 < alphas[i]) and (C > alphas[i]): b = b1
                elif (0 < alphas[j]) and (C > alphas[j]): b = b2
                else: b = (b1 + b2)/2.0
                #统计优化次数
                alphaPairsChanged += 1
                #打印统计信息
                print("第%d次迭代 样本:%d, alpha优化次数:%d" % (iter_num,i,alphaPairsChanged))
        #更新迭代次数
        if (alphaPairsChanged == 0): iter_num += 1
        else: iter_num = 0
        print("迭代次数: %d" % iter_num)
    return b,alphas

"""
函数说明:分类结果可视化

Parameters:
    dataMat - 数据矩阵
    w - 直线法向量
    b - 直线解决
Returns:
    无
"""
def showClassifer(dataMat, w, b):
    #绘制样本点
    data_plus = []                                  #正样本
    data_minus = []                                 #负样本
    for i in range(len(dataMat)):
        if labelMat[i] > 0:
            data_plus.append(dataMat[i])
        else:
            data_minus.append(dataMat[i])
    data_plus_np = np.array(data_plus)              #转换为numpy矩阵
    data_minus_np = np.array(data_minus)            #转换为numpy矩阵
```

```
    plt.scatter(np.transpose(data_plus_np)[0], np.transpose(data_plus_np)[1], s=30, alpha=0.7)   #正样本散点图
    plt.scatter(np.transpose(data_minus_np)[0], np.transpose(data_minus_np)[1], s=30, alpha=0.7) #负样本散点图
    #绘制直线
    x1 = max(dataMat)[0]
    x2 = min(dataMat)[0]
    a1, a2 = w
    b = float(b)
    a1 = float(a1[0])
    a2 = float(a2[0])
    y1, y2 = (-b- a1*x1)/a2, (-b - a1*x2)/a2
    plt.plot([x1, x2], [y1, y2])
    #找出支持向量点
    for i, alpha in enumerate(alphas):
        if abs(alpha) > 0:
            x, y = dataMat[i]
            plt.scatter([x], [y], s=150, c='none', alpha=0.7, linewidth=1.5, edgecolor='red')
    plt.show()


"""
函数说明:计算w

Parameters:
    dataMat - 数据矩阵
    labelMat - 数据标签
    alphas - alphas值
Returns:
    无
"""
def get_w(dataMat, labelMat, alphas):
    alphas, dataMat, labelMat = np.array(alphas), np.array(dataMat), np.array(labelMat)
    w = np.dot((np.tile(labelMat.reshape(1, -1).T, (1, 2)) * dataMat).T, alphas)
    return w.tolist()


if __name__ == '__main__':
    dataMat, labelMat = loadDataSet('testSet.txt')
    b,alphas = smoSimple(dataMat, labelMat, 0.6, 0.001, 40)
    w = get_w(dataMat, labelMat, alphas)
    showClassifer(dataMat, w, b)
```



通过前面的设置进行运行程序并测试时间，我们发现大概是5s左右，虽然看起来不太差，但是不要忘记了这只是一个仅有100个点的小规模数据集而已，这就意味着在更大的数据集上，收敛时间会变得更长，所以我们将通过完整的SMO算法进行加速。

4、加速优化的完整版 Platt SMO 算法

在这两个版本（简化版和完整版）中，实现alpha 的更改和代数运算的优化环节一模一样。在优化过程中，唯一的不同就是 **选择alpha的方式**。完整版的Platt SMO算法应用了一些能够提速的启发方法。

Platt SMO算法是通过一个 **外循环** 来选择第一个alpha值的，并且其选择过程会在两种方式之间进行交替：一种方式是在所有数据集上进行单遍扫描，另一种方式则是在非边界alpha中实现单遍扫描。而所谓非边界alpha指的就是那些不等于边界0或C的alpha值。对整个数据集的扫描相当容易，而实现非边界alpha值的扫描时，首先需要建立这些alpha值的列表，然后再对这个表进行遍历。同时，该步骤会跳过那些已知的不会改变的alpha值。

在选择第一个alpha值后，算法会通过一个 **内循环** 来选择第二个alpha值。在优化过程中，会通过 **最大化步长** 的方式来获得第二个alpha值。在简化版SMO算法中，我们会在选择j 之后计算错误率 Ej。但在这里，我们会建立一个全局的缓存用于保存误差值，并从中选择使得步长或者说 Ei-Ej 最大的alpha 值。

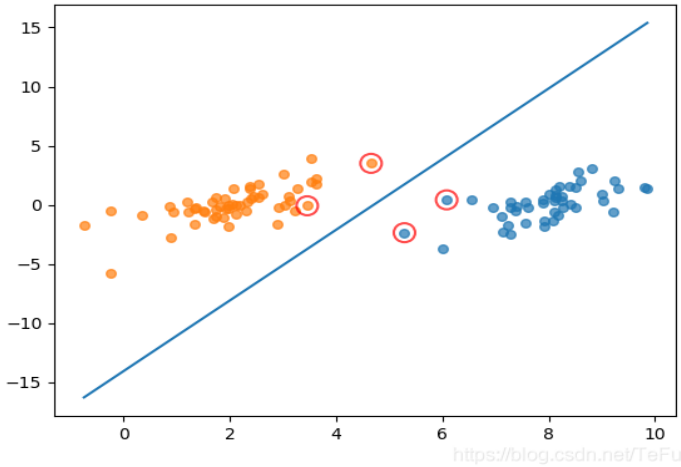
```
import matplotlib.pyplot as plt
import numpy as np
import random

"""
   Parameters:
       dataMatIn - 数据矩阵
       classLabels - 数据标签
       C - 松弛变量
       toler - 容错率
   """
# 数据结构，维护所有需要操作的值（书上说是用于清理代码的数据结构）
class optStruct:
    def __init__(self, dataMatIn, classLabels, C, toler):
        self.X = dataMatIn#数据矩阵
        self.labelMat = classLabels#数据标签
        self.C = C#松弛变量
        self.tol = toler#容错率
        self.m = np.shape(dataMatIn)[0]#数据矩阵行数
        self.alphas = 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
        #根据矩阵行数初始化虎误差缓存，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.eCache = np.mat(np.zeros((self.m,2)))


"""
Parameters:
    fileName - 文件名
Returns:
    dataMat - 数据矩阵
    labelMat - 数据标签
"""
# 读取数据
def loadDataSet(fileName):
    dataMat = []; labelMat = []
    fr = open(fileName)
    for line in fr.readlines():#逐行读取，滤除空格等
        lineArr = line.strip().split('\t')
        dataMat.append([float(lineArr[0]), float(lineArr[1])])#添加数据
        labelMat.append(float(lineArr[2]))#添加标签
    return dataMat,labelMat

"""
Parameters:
    oS - 数据结构
    k - 标号为k的数据
Returns:
    Ek - 标号为k的数据误差
"""
# 计算误差
def calcEk(oS, k):
    fXk = float(np.multiply(oS.alphas,oS.labelMat).T*(oS.X*oS.X[k,:].T) + oS.b)
    Ek = fXk - float(oS.labelMat[k])
    return Ek

"""
Parameters:
    i - alpha_i的索引值
    m - alpha参数个数
Returns:
    j - alpha_j的索引值
"""
# 函数说明:随机选择alpha_j的索引值
```

```python
def selectJrand(i, m):
    j = i#选择一个不等于i的j
    while (j == i):
        j = int(random.uniform(0, m))
    return j

"""
Parameters:
    i - 标号为i的数据的索引值
    oS - 数据结构
    Ei - 标号为i的数据误差
Returns:
    j, maxK - 标号为j或maxK的数据的索引值
    Ej - 标号为j的数据误差
"""
# 内循环启发方式2
def selectJ(i, oS, Ei):
    maxK = -1; maxDeltaE = 0; Ej = 0#初始化
    oS.eCache[i] = [1,Ei]#根据Ei更新误差缓存
    validEcacheList = np.nonzero(oS.eCache[:,0].A)[0]#返回误差不为0的数据的索引值
    if (len(validEcacheList)) > 1:#有不为0的误差
        for k in validEcacheList:#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

"""
Parameters:
    oS - 数据结构
    k - 标号为k的数据的索引值
Returns:
    无
"""
# 计算Ek,并更新误差缓存
def updateEk(oS, k):
    Ek = calcEk(oS, k)#计算Ek
    oS.eCache[k] = [1,Ek]#更新误差缓存

"""
Parameters:
    aj - alpha_j的值
    H - alpha上限
    L - alpha下限
Returns:
    aj - 修剪后的alpah_j的值
"""
# 修剪alpha_j
def clipAlpha(aj,H,L):
    if aj > H:
        aj = H
    if L > aj:
        aj = L
    return aj

"""
Parameters:
    i - 标号为i的数据的索引值
    oS - 数据结构
Returns:
    1 - 有任意一对alpha值发生变化
    0 - 没有任意一对alpha值发生变化或变化太小
"""
# 优化的SMO算法
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.labelMat[i] * Ei < -oS.tol) and (oS.alphas[i] < oS.C)) or\
     ((oS.labelMat[i] * Ei > oS.tol) and (oS.alphas[i] > 0)):
        #使用内循环启发方式2选择alpha_j,并计算Ej
```

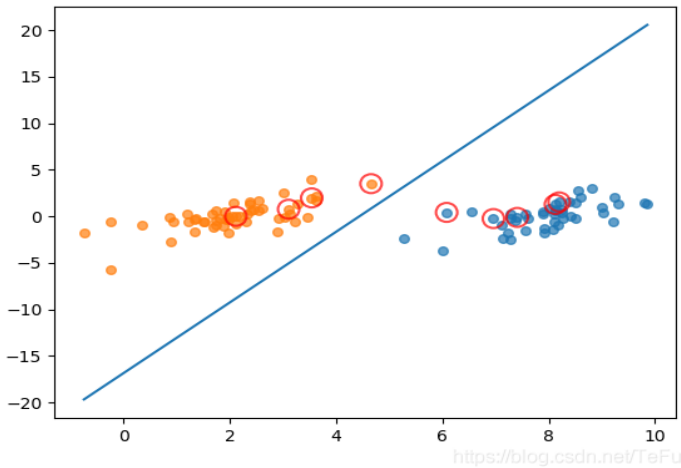
```
        j,Ej = selectJ(i, oS, Ei)
        #保存更新前的aplpha值，使用深拷贝
        alphaIold = oS.alphas[i].copy(); alphaJold = oS.alphas[j].copy();
        #步骤2：计算上下界L和H
        if (oS.labelMat[i] != oS.labelMat[j]):
            L = max(0, oS.alphas[j] - oS.alphas[i])
            H = min(oS.C, oS.C + oS.alphas[j] - oS.alphas[i])
        else:
            L = max(0, oS.alphas[j] + oS.alphas[i] - oS.C)
            H = min(oS.C, oS.alphas[j] + oS.alphas[i])
        if L == H:
            print("L==H")
            return 0
        #步骤3：计算eta
        eta = 2.0 * oS.X[i,:] * oS.X[j,:].T - oS.X[i,:] * oS.X[i,:].T - oS.X[j,:] * oS.X[j,:].T
        if eta >= 0:
            print("eta>=0")
            return 0
        #步骤4：更新alpha_j
        oS.alphas[j] -= oS.labelMat[j] * (Ei - Ej)/eta
        #步骤5：修剪alpha_j
        oS.alphas[j] = clipAlpha(oS.alphas[j],H,L)
        #更新Ej至误差缓存
        updateEk(oS, j)
        if (abs(oS.alphas[j] - alphaJold) < 0.00001):
            print("alpha_j变化太小")
            return 0
        #步骤6：更新alpha_i
        oS.alphas[i] += oS.labelMat[j]*oS.labelMat[i]*(alphaJold - oS.alphas[j])
        #更新Ei至误差缓存
        updateEk(oS, i)
        #步骤7：更新b_1和b_2
        b1 = oS.b - Ei- oS.labelMat[i]*(oS.alphas[i]-alphaIold)*oS.X[i,:]*oS.X[i,:].T - oS.labelMat[j]*(oS.alphas[j]-alphaJold)*
        b2 = oS.b - Ej- oS.labelMat[i]*(oS.alphas[i]-alphaIold)*oS.X[i,:]*oS.X[j,:].T - oS.labelMat[j]*(oS.alphas[j]-alphaJold)*
        #步骤8：根据b_1和b_2更新b
        if (0 < oS.alphas[i]) and (oS.C > oS.alphas[i]): oS.b = b1
        elif (0 < oS.alphas[j]) and (oS.C > oS.alphas[j]): oS.b = b2
        else: oS.b = (b1 + b2)/2.0
        return 1
    else:
        return 0

"""
Parameters:
    dataMatIn - 数据矩阵
    classLabels - 数据标签
    C - 松弛变量
    toler - 容错率
    maxIter - 最大迭代次数
Returns:
    oS.b - SMO算法计算的b
    oS.alphas - SMO算法计算的alphas
"""
# 完整的线性SMO算法
def smoP(dataMatIn, classLabels, C, toler, maxIter):
    oS = optStruct(np.mat(dataMatIn), np.mat(classLabels).transpose(), C, toler)#初始化数据结构
    iter = 0#初始化当前迭代次数
    entireSet = True; alphaPairsChanged = 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 (iter < maxIter) and ((alphaPairsChanged > 0) or (entireSet)):
        alphaPairsChanged = 0
        if entireSet:#遍历整个数据集
            for i in range(oS.m):
                alphaPairsChanged += innerL(i,oS)#使用优化的SMO算法
                print("全样本遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (iter,i,alphaPairsChanged))
            iter += 1
        else:#遍历非边界值
            nonBoundIs = np.nonzero((oS.alphas.A > 0) * (oS.alphas.A < C))[0]#遍历不在边界0和C的alpha
            for i in nonBoundIs:
                alphaPairsChanged += innerL(i,oS)
                print("非边界遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (iter,i,alphaPairsChanged))
            iter += 1
        if entireSet:#遍历一次后改为非边界遍历
```

```
            entireSet = False
        elif (alphaPairsChanged == 0):#如果alpha没有更新,计算全样本遍历
            entireSet = True
        print("迭代次数: %d" % iter)
    return oS.b,oS.alphas#返回SMO算法计算的b和alphas

"""
Parameters:
    dataMat - 数据矩阵
    w - 直线法向量
    b - 直线解决
Returns:
    无
"""
# 分类结果可视化
def showClassifer(dataMat, classLabels, w, b):
    #绘制样本点
    data_plus = []#正样本
    data_minus = []#负样本
    for i in range(len(dataMat)):
        if classLabels[i] > 0:
            data_plus.append(dataMat[i])
        else:
            data_minus.append(dataMat[i])
    data_plus_np = np.array(data_plus)#转换为numpy矩阵
    data_minus_np = np.array(data_minus)#转换为numpy矩阵
    plt.scatter(np.transpose(data_plus_np)[0], np.transpose(data_plus_np)[1], s=30, alpha=0.7)#正样本散点图
    plt.scatter(np.transpose(data_minus_np)[0], np.transpose(data_minus_np)[1], s=30, alpha=0.7)#负样本散点图
    #绘制直线
    x1 = max(dataMat)[0]
    x2 = min(dataMat)[0]
    a1, a2 = w
    b = float(b)
    a1 = float(a1[0])
    a2 = float(a2[0])
    y1, y2 = (-b- a1*x1)/a2, (-b - a1*x2)/a2
    plt.plot([x1, x2], [y1, y2])
    #找出支持向量点
    for i, alpha in enumerate(alphas):
        if abs(alpha) > 0:
            x, y = dataMat[i]
            plt.scatter([x], [y], s=150, c='none', alpha=0.7, linewidth=1.5, edgecolor='red')
    plt.show()

"""
Parameters:
    dataArr - 数据矩阵
    classLabels - 数据标签
    alphas - alphas值
Returns:
    w - 计算得到的w
"""
# 计算w
def calcWs(alphas,dataArr,classLabels):
    X = np.mat(dataArr); labelMat = np.mat(classLabels).transpose()
    m,n = np.shape(X)
    w = np.zeros((n,1))
    for i in range(m):
        w += np.multiply(alphas[i]*labelMat[i],X[i,:].T)
    return w


if __name__ == '__main__':
    dataArr, classLabels = loadDataSet('testSet.txt')
    b, alphas = smoP(dataArr, classLabels, 0.6, 0.001, 40)
    w = calcWs(alphas,dataArr, classLabels)
    showClassifer(dataArr, classLabels, w, b)
```

优化前:



优化后：

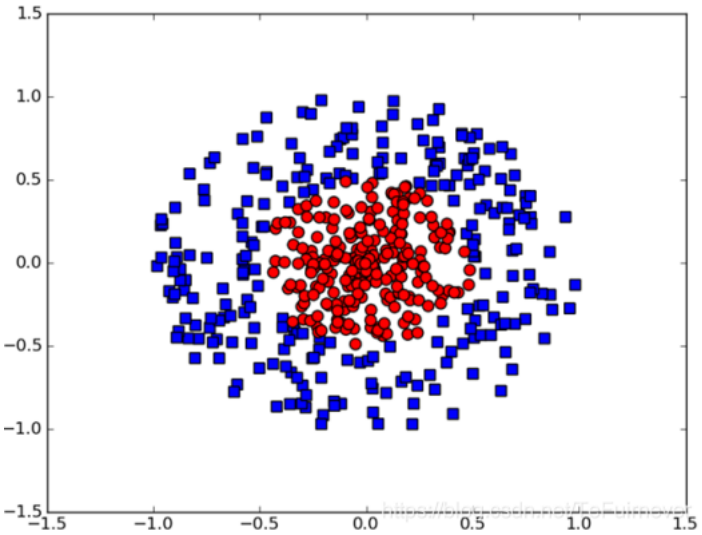


通过下面这个小程序进行计时，发现优化之后的时间相较于之前快了很多，优化前5s左右，优化后2s左右，2倍。

```
import time

start = time.clock()

elapsed1 = (time.clock() - start)
print("Time used:{:.3f}s".format(elapsed1))
```

5、在复杂数据上应用核函数

核函数的目的主要是为了解决非线性分类问题，通过核技巧将低维的非线性特征转化为高维的线性特征，从而可以通过线性模型来解决非线性的分类问题。



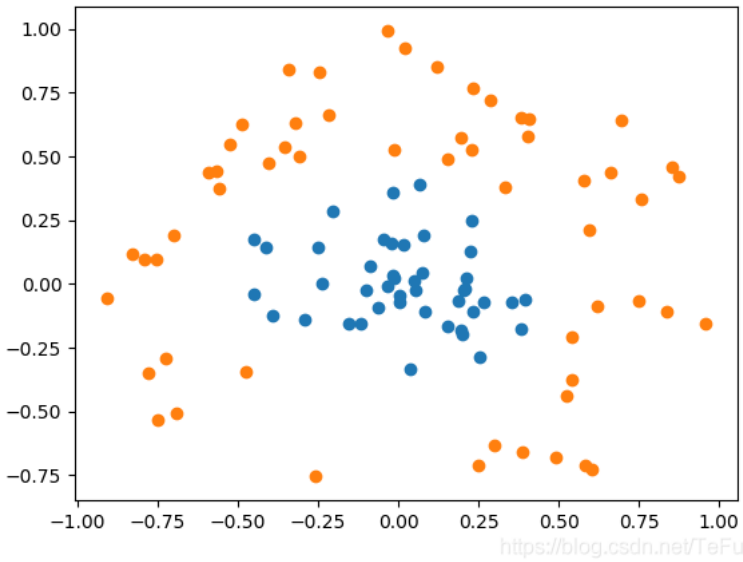
在图中，数据点处于一个圆中，人类的大脑能够意识到这一点。然而，对于分类器而言，它只能识别分类器的结果是大于0还是小于0。如果只在x和y轴构成的坐标系中插入直线进行分类的话，我们并不会得到理想的结果。但是或许可以对圆中的数据进行某种形式的转换，从而得到某些新的变量来表示数据。在这种表示情况下，我们就更容易得到大于0或者小于0的测试结果。在通常情况下，这种映射是通过 **核函数** 来实现的，会将低维特征空间映射到高维空间。

我们可以把核函数想象成一个 **包装器**（wrapper）或者是 **接口**（interface），它能把数据从某个很难处理的形式转换成为另一个较容易处理的形式。如果上述特征空间映射的说法听起来很让人迷糊的话，那么可以将它想象成为另外一种距离计算的方法。距离计算的方法有很多种，核函数一样具有多种类型。经过空间转换之后，我们可以在高维空间中解决线性问题，这也就等价于在低维空间中解决非线性问题。

接下来，我们将使用 testSetRBF.txt 和 testSetRBF2.txt，前者作为训练集，后者作为测试集。

```
# -*-coding:utf-8 -*-
import matplotlib.pyplot as plt
import numpy as np

"""
Parameters:
    fileName - 文件名
Returns:
    dataMat - 数据矩阵
    labelMat - 数据标签
"""
# 读取数据
def loadDataSet(fileName):
    dataMat = []; labelMat = []
    fr = open(fileName)
    for line in fr.readlines():#逐行读取，滤除空格等
        lineArr = line.strip().split('\t')
        dataMat.append([float(lineArr[0]), float(lineArr[1])])#添加数据
        labelMat.append(float(lineArr[2]))#添加标签
    return dataMat,labelMat

"""
数据可视化
Parameters:
    dataMat - 数据矩阵
    labelMat - 数据标签
Returns:
    无
"""
def showDataSet(dataMat, labelMat):
    data_plus = []#正样本
    data_minus = []#负样本
    for i in range(len(dataMat)):
        if labelMat[i] > 0:
            data_plus.append(dataMat[i])
        else:
            data_minus.append(dataMat[i])
    data_plus_np = np.array(data_plus)#转换为numpy矩阵
    data_minus_np = np.array(data_minus)#转换为numpy矩阵
    plt.scatter(np.transpose(data_plus_np)[0], np.transpose(data_plus_np)[1])#正样本散点图
```

```
    plt.scatter(np.transpose(data_minus_np)[0], np.transpose(data_minus_np)[1])#绘制散点图
    plt.show()

if __name__ == '__main__':
    dataArr,labelArr = loadDataSet('testSetRBF.txt')#加载训练集
    showDataSet(dataArr, labelArr)
```



可见，数据明显是线性不可分的。下面我们根据公式，编写核函数，并增加初始化参数kTup用于存储核函数有关的信息，同时我们只要将之前的内积运算变成核函数的运算即可。最后编写testRbf()函数，用于测试。创建svmMLiA.py文件，编写代码如下：

```
# -*-coding:utf-8 -*-
import matplotlib.pyplot as plt
import numpy as np
import random

"""
Parameters:
    dataMatIn - 数据矩阵
    classLabels - 数据标签
    C - 松弛变量
    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
"""
# 数据结构，维护所有需要操作的值
class optStruct:
    def __init__(self, dataMatIn, classLabels, C, toler, kTup):
        self.X = dataMatIn#数据矩阵
        self.labelMat = classLabels#数据标签
        self.C = C#松弛变量
        self.tol = toler#容错率
        self.m = np.shape(dataMatIn)[0]#数据矩阵行数
        self.alphas = 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
        #根据矩阵行数初始化虎误差缓存，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.eCache = np.mat(np.zeros((self.m,2)))
        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)

"""
Parameters:
    X - 数据矩阵
    A - 单个数据的向量
    kTup - 包含核函数信息的元组
Returns:
    K - 计算的核K
"""
# 通过核函数将数据转换更高维的空间
def kernelTrans(X, A, kTup):
```

```
    m,n = np.shape(X)
    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

"""
Parameters:
    fileName - 文件名
Returns:
    dataMat - 数据矩阵
    labelMat - 数据标签
"""
# 读取数据
def loadDataSet(fileName):
    dataMat = []; labelMat = []
    fr = open(fileName)
    for line in fr.readlines():#逐行读取，滤除空格等
        lineArr = line.strip().split('\t')
        dataMat.append([float(lineArr[0]), float(lineArr[1])])#添加数据
        labelMat.append(float(lineArr[2]))#添加标签
    return dataMat,labelMat

"""
Parameters:
    oS - 数据结构
    k - 标号为k的数据
Returns:
    Ek - 标号为k的数据误差
"""
# 计算误差
def calcEk(oS, k):
    fXk = float(np.multiply(oS.alphas,oS.labelMat).T*oS.K[:,k] + oS.b)
    Ek = fXk - float(oS.labelMat[k])
    return Ek

"""
Parameters:
    i - alpha_i的索引值
    m - alpha参数个数
Returns:
    j - alpha_j的索引值
"""
# 函数说明:随机选择alpha_j的索引值
def selectJrand(i, m):
    j = i#选择一个不等于i的j
    while (j == i):
        j = int(random.uniform(0, m))
    return j

"""
Parameters:
    i - 标号为i的数据的索引值
    oS - 数据结构
    Ei - 标号为i的数据误差
Returns:
    j, maxK - 标号为j或maxK的数据的索引值
    Ej - 标号为j的数据误差
"""
# 内循环启发方式2
def selectJ(i, oS, Ei):
    maxK = -1; maxDeltaE = 0; Ej = 0#初始化
    oS.eCache[i] = [1,Ei]#根据Ei更新误差缓存
    validEcacheList = np.nonzero(oS.eCache[:,0].A)[0]#返回误差不为0的数据的索引值
    if (len(validEcacheList)) > 1:#有不为0的误差
        for k in validEcacheList:#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
```

```
    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

"""
Parameters:
    oS - 数据结构
    k - 标号为k的数据的索引值
Returns:
    无
"""
# 计算Ek,并更新误差缓存
def updateEk(oS, k):
    Ek = calcEk(oS, k)#计算Ek
    oS.eCache[k] = [1,Ek]#更新误差缓存

"""
Parameters:
    aj - alpha_j的值
    H - alpha上限
    L - alpha下限
Returns:
    aj - 修剪后的alpah_j的值
"""
# 修剪alpha_j
def clipAlpha(aj,H,L):
    if aj > H:
        aj = H
    if L > aj:
        aj = L
    return aj

"""
Parameters:
    i - 标号为i的数据的索引值
    oS - 数据结构
Returns:
    1 - 有任意一对alpha值发生变化
    0 - 没有任意一对alpha值发生变化或变化太小
"""
# 优化的SMO算法
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.labelMat[i] * Ei < -oS.tol) and (oS.alphas[i] < oS.C)) or\
     ((oS.labelMat[i] * Ei > oS.tol) and (oS.alphas[i] > 0)):
        #使用内循环启发方式2选择alpha_j,并计算Ej
        j,Ej = selectJ(i, oS, Ei)
        #保存更新前的aplpha值，使用深拷贝
        alphaIold = oS.alphas[i].copy(); alphaJold = oS.alphas[j].copy();
        #步骤2：计算上下界L和H
        if (oS.labelMat[i] != oS.labelMat[j]):
            L = max(0, oS.alphas[j] - oS.alphas[i])
            H = min(oS.C, oS.C + oS.alphas[j] - oS.alphas[i])
        else:
            L = max(0, oS.alphas[j] + oS.alphas[i] - oS.C)
            H = min(oS.C, oS.alphas[j] + oS.alphas[i])
        if L == H:
            print("L==H")
            return 0
        #步骤3：计算eta
        eta = 2.0 * oS.K[i,j] - oS.K[i,i] - oS.K[j,j]
        if eta >= 0:
            print("eta>=0")
            return 0
        #步骤4：更新alpha_j
        oS.alphas[j] -= oS.labelMat[j] * (Ei - Ej)/eta
        #步骤5：修剪alpha_j
        oS.alphas[j] = clipAlpha(oS.alphas[j],H,L)
        #更新Ej至误差缓存
        updateEk(oS, j)
        if (abs(oS.alphas[j] - alphaJold) < 0.00001):
            print("alpha_j变化太小")

```

```
            return 0
        #步骤6：更新alpha_i
        oS.alphas[i] += oS.labelMat[j]*oS.labelMat[i]*(alphaJold - oS.alphas[j])
        #更新ei至误差缓存
        updateEk(oS, i)
        #步骤7：更新b_1和b_2
        b1 = oS.b - Ei- oS.labelMat[i]*(oS.alphas[i]-alphaIold)*oS.K[i,i] - oS.labelMat[j]*(oS.alphas[j]-alphaJold)*oS.K[i,j]
        b2 = oS.b - Ej- oS.labelMat[i]*(oS.alphas[i]-alphaIold)*oS.K[i,j]- oS.labelMat[j]*(oS.alphas[j]-alphaJold)*oS.K[j,j]
        #步骤8：根据b_1和b_2更新b
        if (0 < oS.alphas[i]) and (oS.C > oS.alphas[i]): oS.b = b1
        elif (0 < oS.alphas[j]) and (oS.C > oS.alphas[j]): oS.b = b2
        else: oS.b = (b1 + b2)/2.0
        return 1
    else:
        return 0

"""
Parameters:
    dataMatIn - 数据矩阵
    classLabels - 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
Returns:
    oS.b - SMO算法计算的b
    oS.alphas - SMO算法计算的alphas
"""
# 完整的线性SMO算法
def smoP(dataMatIn, classLabels, C, toler, maxIter, kTup = ('lin',0)):
    oS = optStruct(np.mat(dataMatIn), np.mat(classLabels).transpose(), C, toler, kTup)#初始化数据结构
    iter = 0#初始化当前迭代次数
    entireSet = True; alphaPairsChanged = 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 (iter < maxIter) and ((alphaPairsChanged > 0) or (entireSet)):
        alphaPairsChanged = 0
        if entireSet:#遍历整个数据集
            for i in range(oS.m):
                alphaPairsChanged += innerL(i,oS)#使用优化的SMO算法
                print("全样本遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (iter,i,alphaPairsChanged))
            iter += 1
        else:#遍历非边界值
            nonBoundIs = np.nonzero((oS.alphas.A > 0) * (oS.alphas.A < C))[0]#遍历不在边界0和C的alpha
            for i in nonBoundIs:
                alphaPairsChanged += innerL(i,oS)
                print("非边界遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (iter,i,alphaPairsChanged))
            iter += 1
        if entireSet:#遍历一次后改为非边界遍历
            entireSet = False
        elif (alphaPairsChanged == 0):#如果alpha没有更新,计算全样本遍历
            entireSet = True
        print("迭代次数: %d" % iter)
    return oS.b,oS.alphas#返回SMO算法计算的b和alphas

"""
Parameters:
    k1 - 使用高斯核函数的时候表示到达率
Returns:
    无
"""
# 测试函数
def testRbf(k1 = 0.3):
    dataArr,labelArr = loadDataSet('testSetRBF.txt')#加载训练集
    b,alphas = smoP(dataArr, labelArr, 200, 0.0001, 100, ('rbf', k1))#根据训练集计算b和alphas
    datMat = np.mat(dataArr); labelMat = np.mat(labelArr).transpose()
    svInd = np.nonzero(alphas.A > 0)[0]#获得支持向量
    sVs = datMat[svInd]
    labelSV = labelMat[svInd];
    print("支持向量个数:%d" % np.shape(sVs)[0])
    m,n = np.shape(datMat)
    errorCount = 0
    for i in range(m):
        kernelEval = kernelTrans(sVs,datMat[i,:],('rbf', k1))#计算各个点的核
        predict = kernelEval.T * np.multiply(labelSV,alphas[svInd]) + b#根据支持向量的点，计算超平面，返回预测结果

        #返回数组中各元素的正负符号，用1和-1表示，并统计错误个数
```

```
        #返回数组中各元素的正负符号，用1和-1表示，并统计错误个数
        if np.sign(predict) != np.sign(labelArr[i]): errorCount += 1
    print("训练集错误率: %.2f%%" % ((float(errorCount)/m)*100))#打印错误率
    dataArr,labelArr = loadDataSet('testSetRBF2.txt')#加载测试集
    errorCount = 0
    datMat = np.mat(dataArr); labelMat = np.mat(labelArr).transpose()
    m,n = np.shape(datMat)
    for i in range(m):
        kernelEval = kernelTrans(sVs,datMat[i,:],('rbf', k1))#计算各个点的核
        predict=kernelEval.T * np.multiply(labelSV,alphas[svInd]) + b#根据支持向量的点，计算超平面，返回预测结果
        #返回数组中各元素的正负符号，用1和-1表示，并统计错误个数
        if np.sign(predict) != np.sign(labelArr[i]): errorCount += 1
    print("测试集错误率: %.2f%%" % ((float(errorCount)/m)*100))#打印错误率


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

```
>>>
......
......
迭代次数: 3
支持向量个数:25
训练集错误率: 0.00%
测试集错误率: 3.00%
```

可以尝试不同的K1值，经过实验会发现K1越大，过拟合越严重。支持向量的数目存在一个最优值。SVM的优点在于它能对数据进行高效分类。如果支持向量太少，就可能会得到一个很差的决策边界；如果支持向量太多，也就相当于每次都利用整个数据集进行分类，这种分类方法称为k近邻。

6、SVM实现手写数字识别

```
# -*-coding:utf-8 -*-
import matplotlib.pyplot as plt
import numpy as np
import random

"""
数据结构，维护所有需要操作的值
Parameters:
    dataMatIn - 数据矩阵
    classLabels - 数据标签
    C - 松弛变量
    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, dataMatIn, classLabels, C, toler, kTup):
        self.X = dataMatIn#数据矩阵
        self.labelMat = classLabels#数据标签
        self.C = C#松弛变量
        self.tol = toler#容错率
        self.m = np.shape(dataMatIn)[0]#数据矩阵行数
        self.alphas = 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
        #根据矩阵行数初始化虎误差缓存，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.eCache = np.mat(np.zeros((self.m,2)))
        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)

"""
通过核函数将数据转换更高维的空间
Parameters:
    X - 数据矩阵
    A - 单个数据的向量
    kTup - 包含核函数信息的元组
Returns:
    K - 计算的核K
"""
def kernelTrans(X, A, kTup):
```

```
    m,n = np.shape(X)
    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

"""
读取数据
Parameters:
    fileName - 文件名
Returns:
    dataMat - 数据矩阵
    labelMat - 数据标签
"""
def loadDataSet(fileName):
    dataMat = []; labelMat = []
    fr = open(fileName)
    for line in fr.readlines():#逐行读取，滤除空格等
        lineArr = line.strip().split('\t')
        dataMat.append([float(lineArr[0]), float(lineArr[1])])#添加数据
        labelMat.append(float(lineArr[2]))#添加标签
    return dataMat,labelMat

"""
计算误差
Parameters:
    oS - 数据结构
    k - 标号为k的数据
Returns:
    Ek - 标号为k的数据误差
"""
def calcEk(oS, k):
    fXk = float(np.multiply(oS.alphas,oS.labelMat).T*oS.K[:,k] + oS.b)
    Ek = fXk - float(oS.labelMat[k])
    return Ek

"""
函数说明:随机选择alpha_j的索引值

Parameters:
    i - alpha_i的索引值
    m - alpha参数个数
Returns:
    j - alpha_j的索引值
"""
def selectJrand(i, m):
    j = i#选择一个不等于i的j
    while (j == i):
        j = int(random.uniform(0, m))
    return j

"""
 内循环启发方式2
 Parameters:
     i - 标号为i的数据的索引值
     oS - 数据结构
     Ei - 标号为i的数据误差
 Returns:
     j, 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.eCache[i] = [1,Ei]#根据Ei更新误差缓存
    validEcacheList = np.nonzero(oS.eCache[:,0].A)[0]#返回误差不为0的数据的索引值
    if (len(validEcacheList)) > 1:#有不为0的误差
        for k in validEcacheList:#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
```

```
    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

"""
计算Ek,并更新误差缓存
Parameters:
    oS - 数据结构
    k - 标号为k的数据的索引值
Returns:
    无
"""
def updateEk(oS, k):
    Ek = calcEk(oS, k)#计算Ek
    oS.eCache[k] = [1,Ek]#更新误差缓存

"""
修剪alpha_j
Parameters:
    aj - alpha_j的值
    H - alpha上限
    L - alpha下限
Returns:
    aj - 修剪后的alpah_j的值
"""
def clipAlpha(aj,H,L):
    if aj > H:
        aj = H
    if L > aj:
        aj = L
    return aj

"""
优化的SMO算法
Parameters:
    i - 标号为i的数据的索引值
    oS - 数据结构
Returns:
    1 - 有任意一对alpha值发生变化
    0 - 没有任意一对alpha值发生变化或变化太小
"""
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.labelMat[i] * Ei < -oS.tol) and (oS.alphas[i] < oS.C)) or ((oS.labelMat[i] * Ei > oS.tol) and (oS.alphas[i] > 0)):
        #使用内循环启发方式2选择alpha_j,并计算Ej
        j,Ej = selectJ(i, oS, Ei)
        #保存更新前的aplpha值，使用深拷贝
        alphaIold = oS.alphas[i].copy(); alphaJold = oS.alphas[j].copy();
        #步骤2：计算上下界L和H
        if (oS.labelMat[i] != oS.labelMat[j]):
            L = max(0, oS.alphas[j] - oS.alphas[i])
            H = min(oS.C, oS.C + oS.alphas[j] - oS.alphas[i])
        else:
            L = max(0, oS.alphas[j] + oS.alphas[i] - oS.C)
            H = min(oS.C, oS.alphas[j] + oS.alphas[i])
        if L == H:
            print("L==H")
            return 0
        #步骤3：计算eta
        eta = 2.0 * oS.K[i,j] - oS.K[i,i] - oS.K[j,j]
        if eta >= 0:
            print("eta>=0")
            return 0
        #步骤4：更新alpha_j
        oS.alphas[j] -= oS.labelMat[j] * (Ei - Ej)/eta
        #步骤5：修剪alpha_j
        oS.alphas[j] = clipAlpha(oS.alphas[j],H,L)
        #更新Ej至误差缓存
        updateEk(oS, j)
        if (abs(oS.alphas[j] - alphaJold) < 0.00001):
            print("alpha_j变化太小")
            return 0
        #步骤6：更新alpha_i
```

```
        oS.alphas[i] += oS.labelMat[j]*oS.labelMat[i]*(alphaJold - oS.alphas[j])
        #更新ei至误差缓存
        updateEk(oS, i)
        #步骤7：更新b_1和b_2
        b1 = oS.b - Ei- oS.labelMat[i]*(oS.alphas[i]-alphaIold)*oS.K[i,i] - oS.labelMat[j]*(oS.alphas[j]-alphaJold)*oS.K[i,j]
        b2 = oS.b - Ej- oS.labelMat[i]*(oS.alphas[i]-alphaIold)*oS.K[i,j]- oS.labelMat[j]*(oS.alphas[j]-alphaJold)*oS.K[j,j]
        #步骤8：根据b_1和b_2更新b
        if (0 < oS.alphas[i]) and (oS.C > oS.alphas[i]): oS.b = b1
        elif (0 < oS.alphas[j]) and (oS.C > oS.alphas[j]): oS.b = b2
        else: oS.b = (b1 + b2)/2.0
        return 1
    else:
        return 0

"""
完整的线性SMO算法
Parameters:
    dataMatIn - 数据矩阵
    classLabels - 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
Returns:
    oS.b - SMO算法计算的b
    oS.alphas - SMO算法计算的alphas
"""
def smoP(dataMatIn, classLabels, C, toler, maxIter, kTup = ('lin',0)):
    oS = optStruct(np.mat(dataMatIn), np.mat(classLabels).transpose(), C, toler, kTup)#初始化数据结构
    iter = 0#初始化当前迭代次数
    entireSet = True; alphaPairsChanged = 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 (iter < maxIter) and ((alphaPairsChanged > 0) or (entireSet)):
        alphaPairsChanged = 0
        if entireSet:#遍历整个数据集
            for i in range(oS.m):
                alphaPairsChanged += innerL(i,oS)#使用优化的SMO算法
                print("全样本遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (iter,i,alphaPairsChanged))
            iter += 1
        else:#遍历非边界值
            nonBoundIs = np.nonzero((oS.alphas.A > 0) * (oS.alphas.A < C))[0]#遍历不在边界0和C的alpha
            for i in nonBoundIs:
                alphaPairsChanged += innerL(i,oS)
                print("非边界遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (iter,i,alphaPairsChanged))
            iter += 1
        if entireSet:#遍历一次后改为非边界遍历
            entireSet = False
        elif (alphaPairsChanged == 0):#如果alpha没有更新,计算全样本遍历
            entireSet = True
        print("迭代次数: %d" % iter)
    return oS.b,oS.alphas#返回SMO算法计算的b和alphas

"""
将32x32的二进制图像转换为1x1024向量。
Parameters:
    filename - 文件名
Returns:
    returnVect - 返回的二进制图像的1x1024向量
"""
def img2vector(filename):
    returnVect = np.zeros((1,1024))
    fr = open(filename)
    for i in range(32):
        lineStr = fr.readline()
        for j in range(32):
            returnVect[0,32*i+j] = int(lineStr[j])
    return returnVect

"""
加载图片
Parameters:
    dirName - 文件夹的名字
Returns:
    trainingMat - 数据矩阵
    hwLabels - 数据标签
"""
```

```
def loadImages(dirName):
    from os import listdir
    hwLabels = []
    trainingFileList = listdir(dirName)
    m = len(trainingFileList)
    trainingMat = np.zeros((m,1024))
    for i in range(m):
        fileNameStr = trainingFileList[i]
        fileStr = fileNameStr.split('.')[0]
        classNumStr = int(fileStr.split('_')[0])
        if classNumStr == 9: hwLabels.append(-1)
        else: hwLabels.append(1)
        trainingMat[i,:] = img2vector('%s/%s' % (dirName, fileNameStr))
    return trainingMat, hwLabels

"""
Parameters:
    kTup - 包含核函数信息的元组
Returns:
    无
"""
# 测试函数
def testDigits(kTup=('rbf', 10)):
    dataArr,labelArr = loadImages('trainingDigits')
    b,alphas = smoP(dataArr, labelArr, 200, 0.0001, 10, kTup)
    datMat = np.mat(dataArr); labelMat = np.mat(labelArr).transpose()
    svInd = np.nonzero(alphas.A>0)[0]
    sVs=datMat[svInd]
    labelSV = labelMat[svInd];
    print("支持向量个数:%d" % np.shape(sVs)[0])
    m,n = np.shape(datMat)
    errorCount = 0
    for i in range(m):
        kernelEval = kernelTrans(sVs,datMat[i,:],kTup)
        predict=kernelEval.T * np.multiply(labelSV,alphas[svInd]) + b
        if np.sign(predict) != np.sign(labelArr[i]): errorCount += 1
    print("训练集错误率: %.2f%%" % (float(errorCount)/m))
    dataArr,labelArr = loadImages('testDigits')
    errorCount = 0
    datMat = np.mat(dataArr); labelMat = np.mat(labelArr).transpose()
    m,n = np.shape(datMat)
    for i in range(m):
        kernelEval = kernelTrans(sVs,datMat[i,:],kTup)
        predict=kernelEval.T * np.multiply(labelSV,alphas[svInd]) + b
        if np.sign(predict) != np.sign(labelArr[i]): errorCount += 1
    print("测试集错误率: %.2f%%" % (float(errorCount)/m))


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

```
>>>
......
......
迭代次数: 7
支持向量个数:132
训练集错误率: 0.00%
测试集错误率: 0.01%
```

7、Sklearn构建SVM分类器

`Sklearn.svm.SVC` 是一个很好的模型，它是基于libsvm实现的。

```
# -*- coding: UTF-8 -*-
import numpy as np
import operator
from os import listdir
from sklearn.svm import SVC


"""
```

```
Parameters:
    filename - 文件名
Returns:
    returnVect - 返回的二进制图像的1x1024向量
"""
# 将32x32的二进制图像转换为1x1024向量
def img2vector(filename):
    #创建1x1024零向量
    returnVect = np.zeros((1, 1024))
    #打开文件
    fr = open(filename)
    #按行读取
    for i in range(32):
        #读一行数据
        lineStr = fr.readline()
        #每一行的前32个元素依次添加到returnVect中
        for j in range(32):
            returnVect[0, 32*i+j] = int(lineStr[j])
    #返回转换后的1x1024向量
    return returnVect

# 手写数字分类测试
def handwritingClassTest():
    #测试集的Labels
    hwLabels = []
    #返回trainingDigits目录下的文件名
    trainingFileList = listdir('trainingDigits')
    #返回文件夹下文件的个数
    m = len(trainingFileList)
    #初始化训练的Mat矩阵,测试集
    trainingMat = np.zeros((m, 1024))
    #从文件名中解析出训练集的类别
    for i in range(m):
        #获得文件的名字
        fileNameStr = trainingFileList[i]
        #获得分类的数字
        classNumber = int(fileNameStr.split('_')[0])
        #将获得的类别添加到hwLabels中
        hwLabels.append(classNumber)
        #将每一个文件的1x1024数据存储到trainingMat矩阵中
        trainingMat[i,:] = img2vector('trainingDigits/%s' % (fileNameStr))
    clf = SVC(C=200,kernel='rbf')
    clf.fit(trainingMat,hwLabels)
    #返回testDigits目录下的文件列表
    testFileList = listdir('testDigits')
    #错误检测计数
    errorCount = 0.0
    #测试数据的数量
    mTest = len(testFileList)
    #从文件中解析出测试集的类别并进行分类测试
    for i in range(mTest):
        #获得文件的名字
        fileNameStr = testFileList[i]
        #获得分类的数字
        classNumber = int(fileNameStr.split('_')[0])
        #获得测试集的1x1024向量,用于训练
        vectorUnderTest = img2vector('testDigits/%s' % (fileNameStr))
        #获得预测结果
        # classifierResult = classify0(vectorUnderTest, trainingMat, hwLabels, 3)
        classifierResult = clf.predict(vectorUnderTest)
        print("分类返回结果为%d\t真实结果为%d" % (classifierResult, classNumber))
        if(classifierResult != classNumber):
            errorCount += 1.0
    print("总共错了%d个数据\n错误率为%f%%" % (errorCount, errorCount/mTest * 100))


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

```
>>>
......
......
总共错了2个数据
错误率为1.075269%
```

代码和kNN的实现是差不多的，就是换了个分类器而已。

8、sklearn.svm.SVC

sklearn.svm.SVC是一个很好的模型，SVM算法就是通过它实现的，详细的看这个博客——sklearn.svm.SVC()函数解析

9、总结

支持向量机是一种分类器。之所以称为“机”是因为它会产生一个二值决策结果，即它是一种决策“机”。支持向量机的泛化错误率较低，也就是说它具有良好的学习能力，且学到的结果具有很好的推广性。这些优点使得支持向量机十分流行，有些人认为它是监督学习中最好的定式算法。

支持向量机试图通过求解一个二次优化问题来最大化分类间隔。在过去，训练支持向量机常采用非常复杂并且低效的二次规划求解方法。John Platt引入了SMO算法，此算法可以通过每次只优化2个alpha值来加快SVM的训练速度。本章首先讨论了一个简化版本所实现的SMO优化过程，接着给出了完整的Platt SMO算法。相对于简化版而言，完整版算法不仅大大地提高了优化的速度，还使其存在一些进一步提高运行速度的空间。有关这方面的工作，一个经常被引用的参考文献就是“Improvements to Platt’s SMO Algorithm for SVM Classifier Design”。

核方法或者说核技巧会将数据（有时是非线性数据）从一个低维空间映射到一个高维空间，可以将一个在低维空间中的非线性问题转换成高维空间下的线性问题来求解。核方法不止在SVM中适用，还可以用于其他算法中。而其中的径向基函数是一个常用的度量两个向量距离的核函数。支持向量机是一个二类分类器。当用其解决多类问题时，则需要额外的方法对其进行扩展。SVM的效果也对优化参数和所用核函数中的参数敏感。

下一章将通过介绍一个称为boosting的方法来结束我们有关分类的介绍。读者不久就会看到，在boosting和SVM之间存在着许多相似之处。