

机器学习实战教程（九）：支持向量机实战篇之再撕非线性SVM

摘要

上篇文章讲解的是线性SVM的推导过程以及简化版SMO算法的代码实现。本篇文章将讲解SMO算法的优化方法以及非线性SVM。



一、前言

上篇文章讲解的是线性SVM的推导过程以及简化版SMO算法的代码实现。本篇文章将讲解SMO算法的优化方法以及非线性SVM。

本文出现的所有代码，均可在我的github上下载，欢迎Follow、Star：[点击查看](#)

二、SMO算法优化

在几百个点组成的小规模数据集上，简化版SMO算法的运行是没有什么问题的，但是在更大的数据集上的运行速度就会变慢。简化版SMO算法的第二随机的，针对这一问题，我们可以使用启发式选择第二个 α 值，来达到优化效果。

1、启发选择方式

下面这两个公式想必已经不再陌生：

$$\eta = \mathbf{x}_i^T \mathbf{x}_i + \mathbf{x}_j^T \mathbf{x}_j - 2\mathbf{x}_i^T \mathbf{x}_j$$
$$\alpha_j^{new} = \alpha_j^{old} + \frac{y_j(E_i - E_j)}{\eta}$$

在实现SMO算法的时候，先计算 η ，再更新 α_j 。为了加快第二个 α_j 乘子的迭代速度，需要让直线的斜率增大，对于 α_j 的更新公式，其中 η 值没有什么文只能令：

$$\max |E_i - E_j|$$

因此，我们可以明确自己的优化方法了：

- 最外层循环，首先在样本中选择违反KKT条件的一个乘子作为最外层循环，然后用"启发式选择"选择另外一个乘子并进行这两个乘子的优化
- 在非边界乘子中寻找使得 $|E_i - E_j|$ 最大的样本
- 如果没有找到，则从整个样本中随机选择一个样本

接下来，让我们看看完整版SMO算法如何实现。

2、完整版SMO算法

完整版Platt SMO算法是通过一个外循环来选择违反KKT条件的一个乘子，并且其选择过程会在这两种方式之间进行交替：

- 在所有数据集上进行单遍扫描
- 在非边界 α 中实现单遍扫描

非边界 α 指的就是那些不等于边界0或C的 α 值，并且跳过那些已知的不会改变的 α 值。所以我们要先建立这些 α 的列表，用于才能出 α 的更新状态。

在选择第一个 α 值后，算法会通过"启发选择方式"选择第二个 α 值。

3、编写代码

我们首先构建一个仅包含init方法的optStruct类，将其作为一个数据结构来使用，方便我们对于重要数据的维护。代码思路和之前的简化版SMO算法同之处在于增加了优化方法，如果上篇文章已经看懂，我想这个代码会很好理解。创建一个svm-smo.py文件，编写代码如下：

```

1  # -*-coding:utf-8 -*-
2  import matplotlib.pyplot as plt
3  import numpy as np
4  import random
5
6  """
7  Author:
8      Jack Cui
9  Blog:
10     http://blog.csdn.net/c406495762
11  Zhihu:
12     https://www.zhihu.com/people/Jack--Cui/
13  Modify:
14     2017-10-03
15  """
16
17  class optStruct:
18      """
19      数据结构，维护所有需要操作的值
20      Parameters:
21          dataMatIn - 数据矩阵
22          classLabels - 数据标签
23          C - 松弛变量
24          toler - 容错率
25      """
26      def __init__(self, dataMatIn, classLabels, C, toler):
27          self.X = dataMatIn #数据矩阵
28          self.labelMat = classLabels #数据标签
29          self.C = C #松弛变量
30          self.tol = toler #容错率
31          self.m = np.shape(dataMatIn)[0] #数据矩阵行数
32          self.alphas = np.mat(np.zeros((self.m,1))) #根据矩阵行数初始化alpha参数为0
33          self.b = 0 #初始化b参数为0
34          self.eCache = np.mat(np.zeros((self.m,2))) #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。
35
36      def loadDataSet(fileName):
37          """
38          读取数据
39          Parameters:
40              fileName - 文件名
41          Returns:
42              dataMat - 数据矩阵
43              labelMat - 数据标签
44          """
45          dataMat = []; labelMat = []
46          fr = open(fileName)
47          for line in fr.readlines(): #逐行读取，滤除空格等
48              lineArr = line.strip().split('\t')
49              dataMat.append([float(lineArr[0]), float(lineArr[1])]) #添加数据
50              labelMat.append(float(lineArr[2])) #添加标签
51          return dataMat, labelMat
52
53      def calcEk(oS, k):
54          """
55          计算误差
56          Parameters:
57              oS - 数据结构
58              k - 标号为k的数据
59          Returns:
60              Ek - 标号为k的数据误差
61          """
62          fXk = float(np.multiply(oS.alphas, oS.labelMat).T * (oS.X * oS.X[k, :].T) + oS.b)
63          Ek = fXk - float(oS.labelMat[k])
64          return Ek
65
66      def selectJrand(i, m):
67          """
68          函数说明:随机选择alpha_j的索引值
69
70          Parameters:
71              i - alpha_i的索引值
72              m - alpha参数个数
73          Returns:
74              j - alpha_j的索引值
75          """
76          j = i #选择一个不等于i的j
77          while (j == i):
78              j = int(random.uniform(0, m))
79          return j
80
81      def selectJ(i, oS, Ei):
82          """
83          内循环启发方式2
84          Parameters:
85              i - 标号为i的数据的索引值
86              oS - 数据结构
87              Ei - 标号为i的数据误差
88          Returns:
89              j, maxK - 标号为j或maxK的数据的索引值
90              Ej - 标号为j的数据误差
91          """
92          maxK = -1; maxDeltaE = 0; Ej = 0 #初始化
93          oS.eCache[i] = [1, Ei] #根据Ei更新误差缓存
94          validEcacheList = np.nonzero(oS.eCache[:, 0].A)[0] #返回误差不为0的数据的索引值
95          if (len(validEcacheList)) > 1: #有不为0的误差
96              for k in validEcacheList: #遍历，找到最大的Ek
97                  if k == i: continue #不计算i，浪费时间
98                  Ek = calcEk(oS, k) #计算Ek
99                  deltaE = abs(Ei - Ek) #计算|Ei - Ek|

```

```

100         if (deltaE > maxDeltaE):
101             maxK = k; maxDeltaE = deltaE; Ej = Ek
102         return maxK, Ej
103     else:
104         j = selectJrand(i, oS.m)
105         Ej = calcEk(oS, j)
106     return j, Ej
107
108 def updateEk(oS, k):
109     """
110     计算Ek,并更新误差缓存
111     Parameters:
112         oS - 数据结构
113         k - 标号为k的数据的索引值
114     Returns:
115         无
116     """
117     Ek = calcEk(oS, k)
118     oS.eCache[k] = [1,Ek]
119
120
121 def clipAlpha(aj,H,L):
122     """
123     修剪alpha_j
124     Parameters:
125         aj - alpha_j的值
126         H - alpha上限
127         L - alpha下限
128     Returns:
129         aj - 修剪后的alpah_j的值
130     """
131     if aj > H:
132         aj = H
133     if L > aj:
134         aj = L
135     return aj
136
137 def innerL(i, oS):
138     """
139     优化的SMO算法
140     Parameters:
141         i - 标号为i的数据的索引值
142         oS - 数据结构
143     Returns:
144         1 - 有任意一对alpha值发生变化
145         0 - 没有任意一对alpha值发生变化或变化太小
146     """
147     #步骤1: 计算误差Ei
148     Ei = calcEk(oS, i)
149     #优化alpha,设定一定的容错率。
150     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)):
151         #使用内循环启发方式2选择alpha_j,并计算Ej
152         j,Ej = selectJ(i, oS, Ei)
153         #保存更新前的alpha值,使用深拷贝
154         alphaIold = oS.alphas[i].copy(); alphaJold = oS.alphas[j].copy();
155         #步骤2: 计算上下界L和H
156         if (oS.labelMat[i] != oS.labelMat[j]):
157             L = max(0, oS.alphas[j] - oS.alphas[i])
158             H = min(oS.C, oS.C + oS.alphas[j] - oS.alphas[i])
159         else:
160             L = max(0, oS.alphas[j] + oS.alphas[i] - oS.C)
161             H = min(oS.C, oS.alphas[j] + oS.alphas[i])
162         if L == H:
163             print("L==H")
164             return 0
165         #步骤3: 计算eta
166         eta = 2.0 * oS.X[i,:] * oS.X[j,:].T - oS.X[i,:] * oS.X[i,:].T - oS.X[j,:] * oS.X[j,:].T
167         if eta >= 0:
168             print("eta>=0")
169             return 0
170         #步骤4: 更新alpha_j
171         oS.alphas[j] -= oS.labelMat[j] * (Ei - Ej)/eta
172         #步骤5: 修剪alpha_j
173         oS.alphas[j] = clipAlpha(oS.alphas[j],H,L)
174         #更新Ej至误差缓存
175         updateEk(oS, j)
176         if (abs(oS.alphas[j] - alphaJold) < 0.00001):
177             print("alpha_j变化太小")
178             return 0
179         #步骤6: 更新alpha_i
180         oS.alphas[i] += oS.labelMat[j]*oS.labelMat[i]*(alphaJold - oS.alphas[j])
181         #更新Ei至误差缓存
182         updateEk(oS, i)
183         #步骤7: 更新b_1和b_2
184         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)*oS.X[i,:]*oS.X[j,:].T
185         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)*oS.X[j,:]*oS.X[j,:].T
186         #步骤8: 根据b_1和b_2更新b
187         if (0 < oS.alphas[i]) and (oS.C > oS.alphas[i]): oS.b = b1
188         elif (0 < oS.alphas[j]) and (oS.C > oS.alphas[j]): oS.b = b2
189         else: oS.b = (b1 + b2)/2.0
190         return 1
191     else:
192         return 0
193
194 def smoP(dataMatIn, classLabels, C, toler, maxIter):
195     """
196     完整的线性SMO算法
197     Parameters:
198         dataMatIn - 数据矩阵
199         classLabels - 数据标签
200         C - 松弛变量
201         toler - 容错率
202         maxIter - 最大迭代次数
203     Returns:

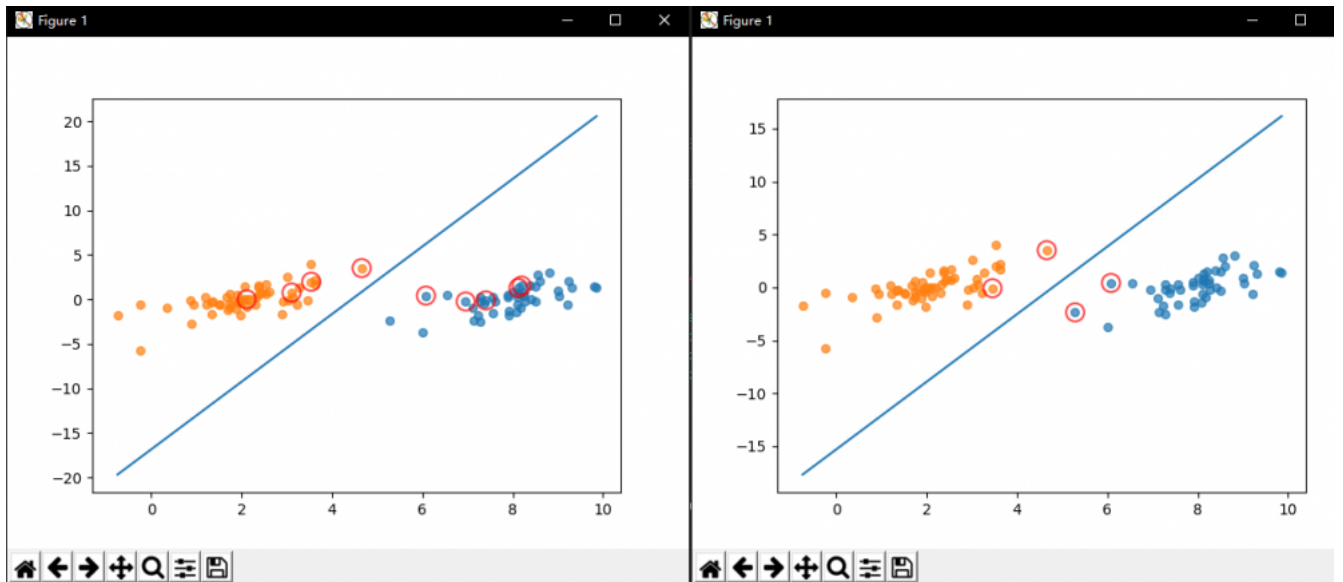
```

```

204         oS.b - SMO算法计算的b
205         oS.alphas - SMO算法计算的alphas
206     """
207     oS = optStruct(np.mat(dataMatIn), np.mat(classLabels).transpose(), C, toler) #初始化数据结构
208     iter = 0 #初始化当前迭代次数
209     entireSet = True; alphaPairsChanged = 0
210     while (iter < maxIter) and ((alphaPairsChanged > 0) or (entireSet)): #遍历整个数据集都alpha也没有更新或者超过最大迭代次数
211         alphaPairsChanged = 0
212         if entireSet: #遍历整个数据集
213             for i in range(oS.m):
214                 alphaPairsChanged += innerL(i,oS) #使用优化的SMO算法
215                 print("全样本遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (iter,i,alphaPairsChanged))
216                 iter += 1
217             else: #遍历非边界值
218                 nonBoundIs = np.nonzero((oS.alphas.A > 0) * (oS.alphas.A < C))[0] #遍历不在边界0和C的alpha
219                 for i in nonBoundIs:
220                     alphaPairsChanged += innerL(i,oS)
221                     print("非边界遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (iter,i,alphaPairsChanged))
222                     iter += 1
223                 if entireSet: #遍历一次后改为非边界遍历
224                     entireSet = False
225                 elif (alphaPairsChanged == 0): #如果alpha没有更新,计算全样本遍历
226                     entireSet = True
227                 print("迭代次数: %d" % iter)
228     return oS.b,oS.alphas #返回SMO算法计算的b和alphas
229
230
231 def showClassifier(dataMat, classLabels, w, b):
232     """
233     分类结果可视化
234     Parameters:
235         dataMat - 数据矩阵
236         w - 直线法向量
237         b - 直线解决
238     Returns:
239         无
240     """
241     #绘制样本点
242     data_plus = [] #正样本
243     data_minus = [] #负样本
244     for i in range(len(dataMat)):
245         if classLabels[i] > 0:
246             data_plus.append(dataMat[i])
247         else:
248             data_minus.append(dataMat[i])
249     data_plus_np = np.array(data_plus) #转换为numpy矩阵
250     data_minus_np = np.array(data_minus) #转换为numpy矩阵
251     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
252     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
253     #绘制直线
254     x1 = max(dataMat)[0]
255     x2 = min(dataMat)[0]
256     a1, a2 = w
257     b = float(b)
258     a1 = float(a1[0])
259     a2 = float(a2[0])
260     y1, y2 = (-b - a1*x1)/a2, (-b - a1*x2)/a2
261     plt.plot([x1, x2], [y1, y2])
262     #找出支持向量点
263     for i, alpha in enumerate(alphas):
264         if abs(alpha) > 0:
265             x, y = dataMat[i]
266             plt.scatter([x], [y], s=150, c='none', alpha=0.7, linewidth=1.5, edgecolor='red')
267     plt.show()
268
269
270 def calcWs(alphas,dataArr,classLabels):
271     """
272     计算w
273     Parameters:
274         dataArr - 数据矩阵
275         classLabels - 数据标签
276         alphas - alphas值
277     Returns:
278         w - 计算得到的w
279     """
280     X = np.mat(dataArr); labelMat = np.mat(classLabels).transpose()
281     m,n = np.shape(X)
282     w = np.zeros((n,1))
283     for i in range(m):
284         w += np.multiply(alphas[i]*labelMat[i],X[i,:].T)
285     return w
286
287 if __name__ == '__main__':
288     dataArr, classLabels = loadDataSet('testSet.txt')
289     b, alphas = smoP(dataArr, classLabels, 0.6, 0.001, 40)
290     w = calcWs(alphas,dataArr, classLabels)
291     showClassifier(dataArr, classLabels, w, b)

```

完整版SMO算法(左图)与简化版SMO算法(右图)运行结果对比如下图所示：



图中画红圈的样本点为支持向量上的点，是满足算法的一种解。完整版SMO算法覆盖整个数据集进行计算，而简化版SMO算法是随机选择的。可以看到SMO算法选出的支持向量样点更多，更接近理想的分隔超平面。

对比两种算法的运算时间，我的测试结果是**完整版SMO算法的速度比简化版SMO算法的速度快6倍左右**。

其实，优化方法不仅仅是简单的启发式选择，还有其他优化方法，SMO算法速度还可以进一步提高。但是鉴于文章进度，这里不再进行展开。感兴趣移步这里进行理论学习：[点我查看](#)

三、非线性SVM

1、核技巧

我们已经了解到，SVM如何处理线性可分的情况，而对于非线性的情况，SVM的处理方式就是选择一个核函数。简而言之：在线性不可分的情况下，事先选择的非线性映射（核函数）将输入变量映射到一个高维特征空间，将其变成在高维空间线性可分，在这个高维空间中构造最优分类超平面。

根据上篇文章，线性可分的情况下，可知最终的超平面方程为：

$$f(\mathbf{x}) = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i^T \mathbf{x} + b$$

将上述公式用内积来表示：

$$f(\mathbf{x}) = \sum_{i=1}^n \alpha_i y_i \langle \mathbf{x}_i, \mathbf{x} \rangle + b$$

对于线性不可分，我们使用一个非线性映射，将数据映射到特征空间，在特征空间中使用线性学习器，分类函数变形如下：

$$f(\mathbf{x}) = \sum_{i=1}^n \alpha_i y_i \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}) \rangle + b$$

其中 ϕ 从输入空间(X)到某个特征空间(F)的映射，这意味着建立非线性学习器分为两步：

- 首先使用一个非线性映射将数据变换到一个特征空间F；
- 然后在特征空间使用线性学习器分类。

如果有一种方法可以在**特征空间中直接计算内积** $\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}) \rangle$ ，就像在原始输入点的函数中一样，就有可能将两个步骤融合到一起建立一个分线性的**直接计算的方法称为核函数方法**。

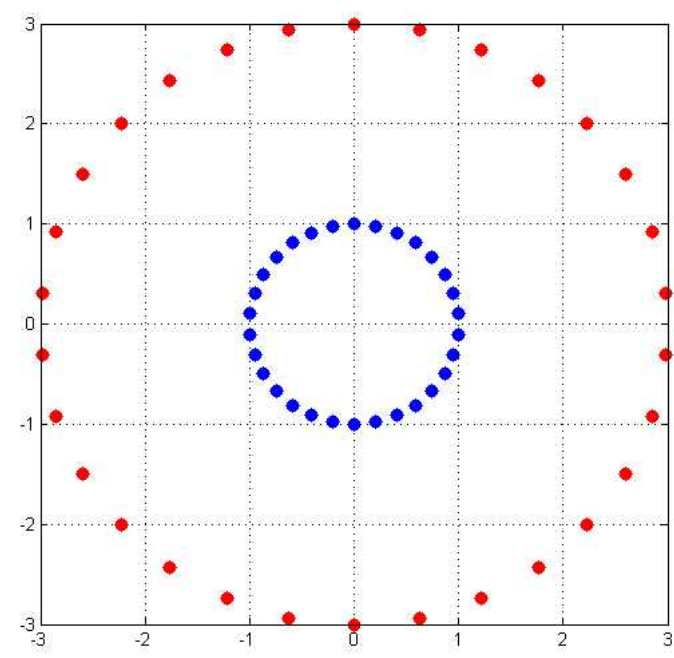
这里直接给出一个定义：核是一个函数 k ，对所有 $\mathbf{x}, \mathbf{z} \in X$ ，满足 $k(\mathbf{x}, \mathbf{z}) = \langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle$ ，这里 $\phi(\cdot)$ 是从原始输入空间X到内积空间F的映射。

简而言之：如果不是用核技术，就会先计算线性映射 $\phi(\mathbf{x}_1)$ 和 $\phi(\mathbf{x}_2)$ ，然后计算它们的内积，使用了核技术之后，先把 $\phi(\mathbf{x}_1)$ 和 $\phi(\mathbf{x}_2)$ 的一般表达式 $\langle \phi(\mathbf{x}_1), \phi(\mathbf{x}_2) \rangle = k(\langle \phi(\mathbf{x}_1), \phi(\mathbf{x}_2) \rangle)$ 计算出来，这里的 $\langle \cdot, \cdot \rangle$ 表示内积， $k(\cdot, \cdot)$ 就是对应的核函数，这个表达式往往非常简单，所以计算非常方便。

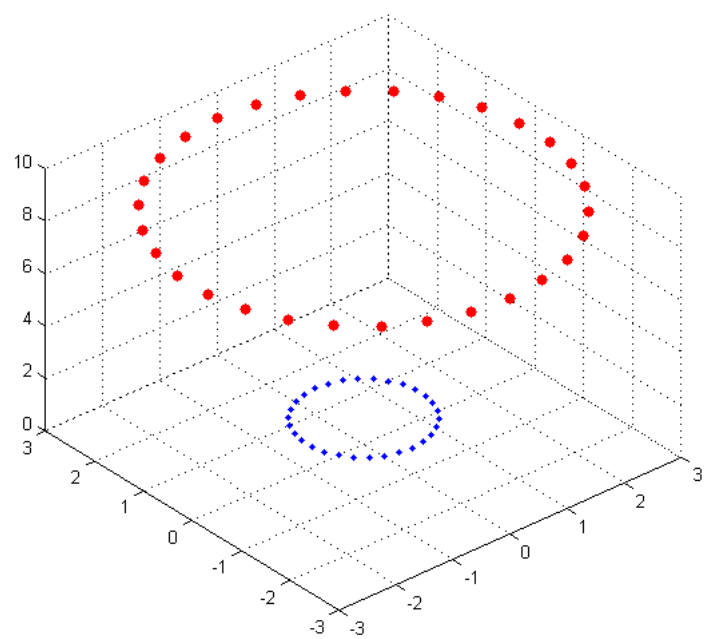
这种将内积替换成核函数的方式被称为**核技巧(kernel trick)**。

2、非线性数据处理

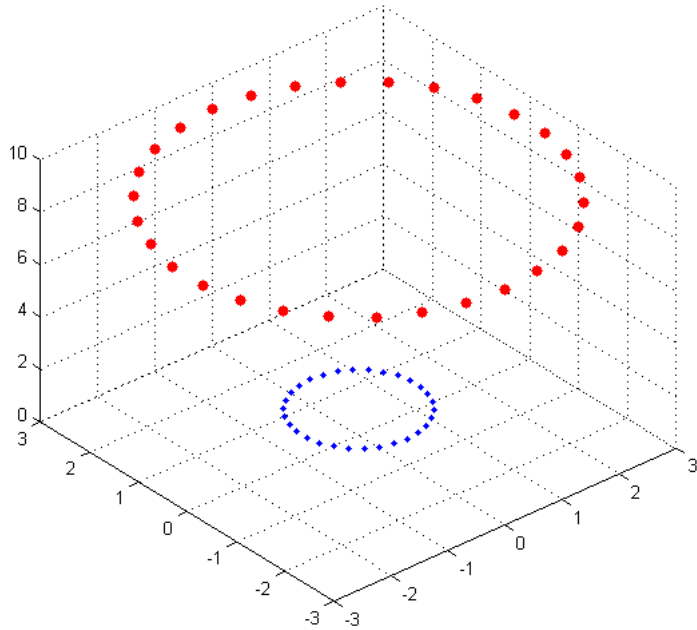
已经知道了核技巧是什么，但是为什么要这样做呢？我们先举一个简单的例子，进行说明。假设二维平面x-y上存在若干点，其中点集A服从 $\{x,y|x^2+y^2=9\}$ 集B服从 $\{x,y|x^2+y^2=9\}$ ，那么这些点在二维平面上的分布是这样的：



蓝色的是点集A，红色的是点集B，他们在xy平面上并不能线性可分，即用一条直线分割（虽然肉眼是可以识别的）。采用映射 $(x,y) \rightarrow (x,y,x^2+y^2)$ 间的点的分布为：

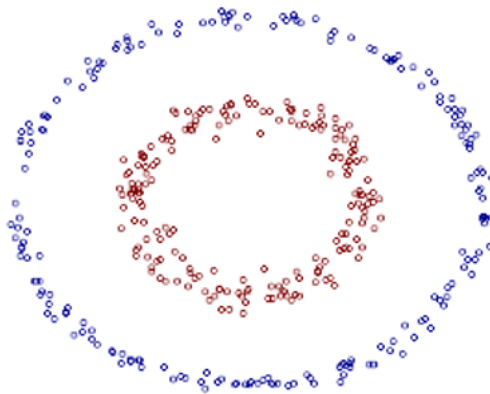


可见红色和蓝色的点被映射到了不同的平面，在更高维空间中是线性可分的（用一个平面去分割）。
上述例子中的样本点的分布遵循圆的分布。继续推广到椭圆的一般样本形式：



可见红色和蓝色的点被映射到了不同的平面，在更高维空间中是线性可分的（用一个平面去分割）。

上述例子中的样本点的分布遵循圆的分布。继续推广到椭圆的一般样本形式：



上图的两类数据分布为两个椭圆的形状，这样的数据本身就是不可分的。不难发现，这两个半径不同的椭圆是加上了少量的噪音生成得到的。所以，应该也是一个椭圆，而不是一个直线。如果用 X_1 和 X_2 来表示这个二维平面的两个坐标的话，我们知道这个分界椭圆可以写为：

$$a_1X_1 + a_2X_1^2 + a_3X_2 + a_4X_2^2 + a_5X_1X_2 + a_6 = 0$$

这个方程就是高中学过的椭圆一般方程。注意上面的形式，如果我们构造另外一个五维的空间，其中五个坐标的值分别为：

$$Z_1 = X_1, Z_2 = X_1^2, Z_3 = X_2, Z_4 = X_2^2, Z_5 = X_1X_2$$

那么，显然我们可以将这个分界的椭圆方程写成如下形式：

$$\sum_{i=1}^5 a_i Z_i + a_6 = 0$$

这个关于新的坐标 Z_1, Z_2, Z_3, Z_4, Z_5 的方程，就是一个超平面方程，它的维度是5。也就是说，如果我们做一个映射 ϕ ：二维 \rightarrow 五维，将 X_1, X_2 按照上 Z_1, Z_2, \dots, Z_5 ，那么在新的空间中原来的数据将变成线性可分的，从而使用之前我们推导的线性分类算法就可以进行了。

我们举个简单的计算例子，现在假设已知的映射函数为：

$$\phi((x_1, x_2)) = (\sqrt{2}x_1, x_1^2, \sqrt{2}x_2, x_2^2, \sqrt{2}x_1x_2, 1)$$

这个是一个从2维映射到5维的例子。如果没有使用核函数，根据上一小节的介绍，我们需要先计算映射后的结果，然后再进行内积运算。那么对于两 (x_1, x_2) 和 $a_2=(y_1, y_2)$ 有：

$$\begin{aligned} \langle \phi((x_1, x_2), \phi(y_1, y_2)) \rangle \\ = 2x_1y_1 + x_1^2y_1^2 + 2x_2y_2 + x_2^2y_2^2 + 2x_1x_2y_1y_2 + 1 \end{aligned}$$

另外，如果我们不进行映射计算，直接运算下面的公式：

$$(\langle x_1, x_2 \rangle + 1)^2 = 2x_1y_1 + x_1^2y_1^2 + 2x_2y_2 + x_2^2y_2^2 + 2x_1x_2y_1y_2 + 1$$

你会发现，这两个公式的计算结果是相同的。区别在于什么呢？

- 一个是根据映射函数，映射到高维空间中，然后再根据内积的公式进行计算，计算量大；
- 另一个则直接在原来的低维空间中进行计算，而不需要显式地写出映射后的结果，计算量小。

其实，在这个例子中，核函数就是：

$$k(x_1, x_2) = (\langle x_1, x_2 \rangle + 1)^2$$

我们通过 $k(x_1, x_2)$ 的低维运算得到了先映射再内积的高维运算的结果，这就是核函数的神奇之处，它有效减少了我们的计算量。在这个例子中，我们对 ϕ 做映射，选择的新的空间是原始空间的所以一阶和二阶的组合，得到了5维的新空间；如果原始空间是3维的，那么我们会得到19维的新空间，这个数目是的。如果我们使用 $\phi(\cdot)$ 做映射计算，难度非常大，而且如果遇到无穷维的情况，就根本无从计算了。所以使用核函数进行计算是非常有必要的。

3、核技巧的实现

通过核技巧的转变，我们的分类函数变为：

$$f(x) = \sum_{i=1}^n \alpha_i y_i \kappa(x_i, x) + b$$

我们的对偶问题变成了：

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^n \alpha_i - \frac{1}{2} \alpha_i \alpha_j y_i y_j \kappa(x_i, x_j) \\ \text{s.t.} \quad & \alpha_i \geq 0, i = 1, 2, \dots, n \\ & \sum_{i=1}^n \alpha_i y_i = 0 \end{aligned}$$

这样，我们就避开了高纬度空间中的计算。当然，我们刚刚的例子是非常简单的，我们可以手动构造出来对应映射的核函数出来，如果对于任意一个 ϕ 对应的核函数就很困难了。因此，通常，人们会从一些常用的核函数中进行选择，根据问题和数据的不同，选择不同的参数，得到不同的核函数。接下来一个非常流行的核函数，那就是径向基核函数。

径向基核函数是SVM中常用的一个核函数。径向基核函数采用向量作为自变量的函数，能够基于向量举例运算输出一个标量。径向基核函数的高斯版本下：

$$\kappa(x_1, x_2) = \exp \left\{ -\frac{\|x_1 - x_2\|^2}{2\sigma^2} \right\}$$

其中， σ 是用户自定义的用于确定到达率(reach)或者说函数值跌落到0的速度参数。上述高斯核函数将数据从原始空间映射到无穷维空间。关于无穷维不必太担心。高斯核函数只是一个常用的核函数，使用者并不需要确切地理解数据到底是如何表现的，而且使用高斯核函数还会得到一个理想的结果。如果 σ 选得很大，高次特征上的权重实际上衰减得非常快，所以实际上（数值上近似一下）相当于一个低维的子空间；反过来，如果 σ 选得很小，则可以将任意的数据映射到高维空间——当然，这并不一定是好事，因为随之而来的可能是非常严重的过拟合问题。不过，总的来说，通过调控参数 σ ，高斯核实际上具有相当高的灵活性，它是核函数之一。

四、编程实现非线性SVM

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。数据集下载地址：[点我查看](#)

1、可视化数据集

我们先编写程序简单看下数据集：

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

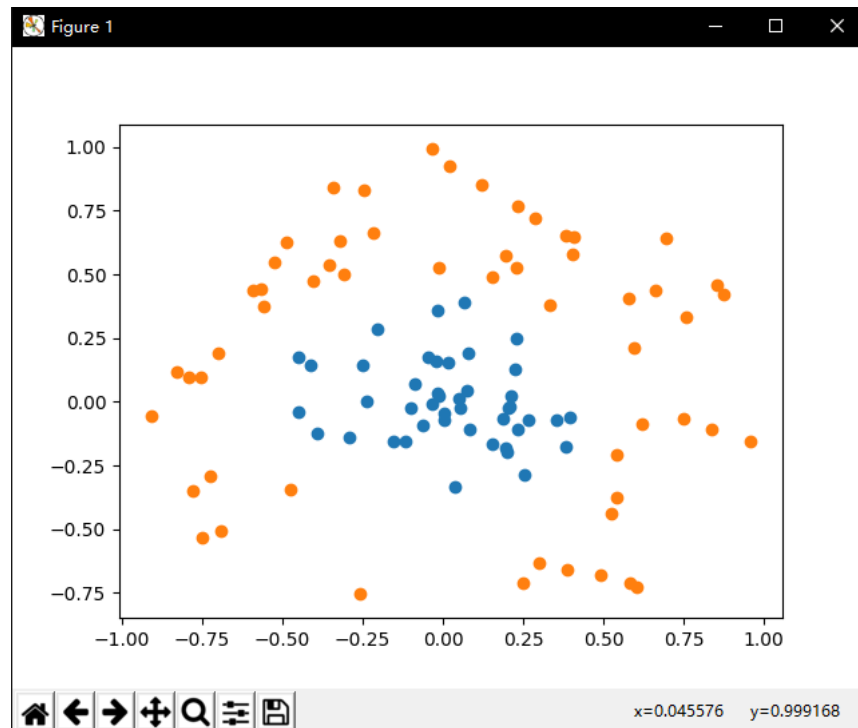


```

4
5 def showDataSet(dataMat, labelMat):
6     """
7     数据可视化
8     Parameters:
9         dataMat - 数据矩阵
10        labelMat - 数据标签
11    Returns:
12        无
13    """
14    data_plus = []
15    data_minus = []
16    for i in range(len(dataMat)):
17        if labelMat[i] > 0:
18            data_plus.append(dataMat[i])
19        else:
20            data_minus.append(dataMat[i])
21    data_plus_np = np.array(data_plus)
22    data_minus_np = np.array(data_minus)
23    plt.scatter(np.transpose(data_plus_np)[0], np.transpose(data_plus_np)[1])
24    plt.scatter(np.transpose(data_minus_np)[0], np.transpose(data_minus_np)[1])
25    plt.show()
26
27 if __name__ == '__main__':
28     dataArr, labelArr = loadDataSet('testSetRbf.txt')
29     showDataSet(dataArr, labelArr)

```

程序运行结果：



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：

```

1 # -*-coding:utf-8 -*-
2 import matplotlib.pyplot as plt
3 import numpy as np
4 import random
5
6 """
7 Author:
8     Jack Cui
9 Blog:
10    http://blog.csdn.net/c406495762
11 Zhihu:
12    https://www.zhihu.com/people/Jack--Cui/
13 Modify:
14    2017-10-03
15 """
16
17 class optStruct:
18     """
19     数据结构，维护所有需要操作的值
20     Parameters:
21         dataMatIn - 数据矩阵
22         classLabels - 数据标签
23         C - 松弛变量
24         toler - 容错率
25         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
26     """
27     def __init__(self, dataMatIn, classLabels, C, toler, kTup):
28         self.X = dataMatIn
29         self.labelMat = classLabels
30         self.C = C
31         self.tol = toler
32         self.m = np.shape(dataMatIn)[0]

```

```

33         self.alphas = np.mat(np.zeros((self.m,1)))          #根据矩阵行数初始化alpha参数为0
34         self.b = 0                                           #初始化b参数为0
35         self.eCache = 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，第二列为实际的误差E的值。
36         self.K = np.mat(np.zeros((self.m,self.m)))          #初始化核K
37         for i in range(self.m):                              #计算所有数据的核K
38             self.K[:,i] = kernelTrans(self.X, self.X[i,:], kTup)
39
40     def kernelTrans(X, A, kTup):
41         """
42         通过核函数将数据转换更高维的空间
43         Parameters:
44             X - 数据矩阵
45             A - 单个数据的向量
46             kTup - 包含核函数信息的元组
47         Returns:
48             K - 计算的核K
49         """
50         m,n = np.shape(X)
51         K = np.mat(np.zeros((m,1)))
52         if kTup[0] == 'lin': K = X * A.T                      #线性核函数,只进行内积。
53         elif kTup[0] == 'rbf':                                #高斯核函数,根据高斯核函数公式进行计算
54             for j in range(m):
55                 deltaRow = X[j,:] - A
56                 K[j] = deltaRow*deltaRow.T
57             K = np.exp(K/(-1*kTup[1]**2))                    #计算高斯核K
58         else: raise NameError('核函数无法识别')
59         return K                                              #返回计算的核K
60
61     def loadDataSet(fileName):
62         """
63         读取数据
64         Parameters:
65             fileName - 文件名
66         Returns:
67             dataMat - 数据矩阵
68             labelMat - 数据标签
69         """
70         dataMat = []; labelMat = []
71         fr = open(fileName)
72         for line in fr.readlines():                          #逐行读取，滤除空格等
73             lineArr = line.strip().split('\t')
74             dataMat.append([float(lineArr[0]), float(lineArr[1])]) #添加数据
75             labelMat.append(float(lineArr[2]))                #添加标签
76         return dataMat,labelMat
77
78     def calcEk(oS, k):
79         """
80         计算误差
81         Parameters:
82             oS - 数据结构
83             k - 标号为k的数据
84         Returns:
85             Ek - 标号为k的数据误差
86         """
87         fXk = float(np.multiply(oS.alphas,oS.labelMat).T*oS.K[:,k] + oS.b)
88         Ek = fXk - float(oS.labelMat[k])
89         return Ek
90
91     def selectJrand(i, m):
92         """
93         函数说明:随机选择alpha_j的索引值
94
95         Parameters:
96             i - alpha_i的索引值
97             m - alpha参数个数
98         Returns:
99             j - alpha_j的索引值
100         """
101         j = i                                                  #选择一个不等于i的j
102         while (j == i):
103             j = int(random.uniform(0, m))
104         return j
105
106     def selectJ(i, oS, Ei):
107         """
108         内循环启发方式2
109         Parameters:
110             i - 标号为i的数据的索引值
111             oS - 数据结构
112             Ei - 标号为i的数据误差
113         Returns:
114             j, maxK - 标号为j或maxK的数据的索引值
115             Ej - 标号为j的数据误差
116         """
117         maxK = -1; maxDeltaE = 0; Ej = 0                      #初始化
118         oS.eCache[i] = [1,Ei]                                #根据Ei更新误差缓存
119         validEcacheList = np.nonzero(oS.eCache[:,0].A)[0]    #返回误差不为0的数据的索引值
120         if (len(validEcacheList)) > 1:
121             for k in validEcacheList:
122                 if k == i: continue                            #不为0的误差
123                 Ek = calcEk(oS, k)                             #遍历,找到最大的Ek
124                 deltaE = abs(Ei - Ek)                          #不计算i,浪费时间
125                 if (deltaE > maxDeltaE):                        #计算Ek
126                     maxK = k; maxDeltaE = deltaE; Ej = Ek     #计算|Ei-Ek|
127             return maxK, Ej                                    #找到maxDeltaE
128         else:
129             j = selectJrand(i, oS.m)                          #没有不为0的误差
130             Ej = calcEk(oS, j)                                  #随机选择alpha_j的索引值
131             return j, Ej                                       #计算Ej
132
133     def updateEk(oS, k):
134         """
135         计算Ek,并更新误差缓存
136         Parameters:

```

```

137         oS - 数据结构
138         k - 标号为k的数据的索引值
139     Returns:
140         无
141     """
142     Ek = calcEk(oS, k) #计算Ek
143     oS.eCache[k] = [1, Ek] #更新误差缓存
144
145 def clipAlpha(aj, H, L):
146     """
147     修剪alpha_j
148     Parameters:
149         aj - alpha_j的值
150         H - alpha上限
151         L - alpha下限
152     Returns:
153         aj - 修剪后的alpha_j的值
154     """
155     if aj > H:
156         aj = H
157     if L > aj:
158         aj = L
159     return aj
160
161 def innerL(i, oS):
162     """
163     优化的SMO算法
164     Parameters:
165         i - 标号为i的数据的索引值
166         oS - 数据结构
167     Returns:
168         1 - 有任意一对alpha值发生变化
169         0 - 没有任意一对alpha值发生变化或变化太小
170     """
171     #步骤1: 计算误差Ei
172     Ei = calcEk(oS, i)
173     #优化alpha, 设定一定的容错率。
174     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)):
175         #使用内循环启发方式2选择alpha_j, 并计算Ej
176         j, Ej = selectJ(i, oS, Ei)
177         #保存更新前的alpha值, 使用深拷贝
178         alphaIold = oS.alphas[i].copy(); alphaJold = oS.alphas[j].copy();
179         #步骤2: 计算上下界L和H
180         if (oS.labelMat[i] != oS.labelMat[j]):
181             L = max(0, oS.alphas[j] - oS.alphas[i])
182             H = min(oS.C, oS.C + oS.alphas[j] - oS.alphas[i])
183         else:
184             L = max(0, oS.alphas[j] + oS.alphas[i] - oS.C)
185             H = min(oS.C, oS.alphas[j] + oS.alphas[i])
186         if L == H:
187             print("L==H")
188             return 0
189         #步骤3: 计算eta
190         eta = 2.0 * oS.K[i, j] - oS.K[i, i] - oS.K[j, j]
191         if eta >= 0:
192             print("eta>=0")
193             return 0
194         #步骤4: 更新alpha_j
195         oS.alphas[j] -= oS.labelMat[j] * (Ei - Ej)/eta
196         #步骤5: 修剪alpha_j
197         oS.alphas[j] = clipAlpha(oS.alphas[j], H, L)
198         #更新Ej至误差缓存
199         updateEk(oS, j)
200         if (abs(oS.alphas[j] - alphaJold) < 0.00001):
201             print("alpha_j变化太小")
202             return 0
203         #步骤6: 更新alpha_i
204         oS.alphas[i] += oS.labelMat[j]*oS.labelMat[i]*(alphaJold - oS.alphas[j])
205         #更新Ei至误差缓存
206         updateEk(oS, i)
207         #步骤7: 更新b_1和b_2
208         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]
209         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]
210         #步骤8: 根据b_1和b_2更新b
211         if (0 < oS.alphas[i]) and (oS.C > oS.alphas[i]): oS.b = b1
212         elif (0 < oS.alphas[j]) and (oS.C > oS.alphas[j]): oS.b = b2
213         else: oS.b = (b1 + b2)/2.0
214         return 1
215     else:
216         return 0
217
218 def smoP(dataMatIn, classLabels, C, toler, maxIter, kTup = ('lin', 0)):
219     """
220     完整的线性SMO算法
221     Parameters:
222         dataMatIn - 数据矩阵
223         classLabels - 数据标签
224         C - 松弛变量
225         toler - 容错率
226         maxIter - 最大迭代次数
227         kTup - 包含核函数信息的元组
228     Returns:
229         oS.b - SMO算法计算的b
230         oS.alphas - SMO算法计算的alphas
231     """
232     oS = optStruct(np.mat(dataMatIn), np.mat(classLabels).transpose(), C, toler, kTup) #初始化数据结构
233     iter = 0 #初始化当前迭代次数
234     entireSet = True; alphaPairsChanged = 0
235     while (iter < maxIter) and ((alphaPairsChanged > 0) or (entireSet)): #遍历整个数据集都alpha也没有更新或者超过最大迭代次数
236         alphaPairsChanged = 0
237         if entireSet: #遍历整个数据集
238             for i in range(oS.m):
239                 alphaPairsChanged += innerL(i, oS) #使用优化的SMO算法
240                 print("全样本遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (iter, i, alphaPairsChanged))

```

```

241     iter += 1
242 else:
243     nonBoundIs = np.nonzero((oS.alphas.A > 0) * (oS.alphas.A < C))[0]
244     for i in nonBoundIs:
245         alphaPairsChanged += innerL(i,oS)
246         print("非边界遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (iter,i,alphaPairsChanged))
247     iter += 1
248     if entireSet:
249         entireSet = False
250     elif (alphaPairsChanged == 0):
251         entireSet = True
252     print("迭代次数: %d" % iter)
253     return oS.b,oS.alphas
254
255 def testRbf(k1 = 1.3):
256     """
257     测试函数
258     Parameters:
259         k1 - 使用高斯核函数的时候表示到达率
260     Returns:
261         无
262     """
263     dataArr,labelArr = loadDataSet('testSetRBF.txt')
264     b,alphas = smOP(dataArr, labelArr, 200, 0.0001, 100, ('rbf', k1))
265     datMat = np.mat(dataArr); labelMat = np.mat(labelArr).transpose()
266     svInd = np.nonzero(alphas.A > 0)[0]
267     sVs = datMat[svInd]
268     labelSV = labelMat[svInd];
269     print("支持向量个数:%d" % np.shape(sVs)[0])
270     m,n = np.shape(datMat)
271     errorCount = 0
272     for i in range(m):
273         kernelEval = kernelTrans(sVs,datMat[i,:],('rbf', k1))
274         predict = kernelEval.T * np.multiply(labelSV,alphas[svInd]) + b
275         if np.sign(predict) != np.sign(labelArr[i]): errorCount += 1
276     print("训练集错误率: %.2f%%" % ((float(errorCount)/m)*100))
277     dataArr,labelArr = loadDataSet('testSetRBF2.txt')
278     errorCount = 0
279     datMat = np.mat(dataArr); labelMat = np.mat(labelArr).transpose()
280     m,n = np.shape(datMat)
281     for i in range(m):
282         kernelEval = kernelTrans(sVs,datMat[i,:],('rbf', k1))
283         predict=kernelEval.T * np.multiply(labelSV,alphas[svInd]) + b
284         if np.sign(predict) != np.sign(labelArr[i]): errorCount += 1
285     print("测试集错误率: %.2f%%" % ((float(errorCount)/m)*100))
286
287 def showDataSet(dataMat, labelMat):
288     """
289     数据可视化
290     Parameters:
291         dataMat - 数据矩阵
292         labelMat - 数据标签
293     Returns:
294         无
295     """
296     data_plus = []
297     data_minus = []
298     for i in range(len(dataMat)):
299         if labelMat[i] > 0:
300             data_plus.append(dataMat[i])
301         else:
302             data_minus.append(dataMat[i])
303     data_plus_np = np.array(data_plus)
304     data_minus_np = np.array(data_minus)
305     plt.scatter(np.transpose(data_plus_np)[0], np.transpose(data_plus_np)[1])
306     plt.scatter(np.transpose(data_minus_np)[0], np.transpose(data_minus_np)[1])
307     plt.show()
308
309 if __name__ == '__main__':
310     testRbf()

```

运行结果如下图所示：

```

309     plt.scatter(np.transpose(data_minus_np)[0], np.transpose(data_minus_np)[1])
310     plt.show()
311
312 if __name__ == '__main__':
313     testRbf()

```

```

alpha_j变化太小
全样本遍历:第5次迭代 样本:87, alpha优化次数:0
全样本遍历:第5次迭代 样本:88, alpha优化次数:0
全样本遍历:第5次迭代 样本:89, alpha优化次数:0
全样本遍历:第5次迭代 样本:90, alpha优化次数:0
全样本遍历:第5次迭代 样本:91, alpha优化次数:0
L==H
全样本遍历:第5次迭代 样本:92, alpha优化次数:0
全样本遍历:第5次迭代 样本:93, alpha优化次数:0
全样本遍历:第5次迭代 样本:94, alpha优化次数:0
全样本遍历:第5次迭代 样本:95, alpha优化次数:0
全样本遍历:第5次迭代 样本:96, alpha优化次数:0
全样本遍历:第5次迭代 样本:97, alpha优化次数:0
全样本遍历:第5次迭代 样本:98, alpha优化次数:0
全样本遍历:第5次迭代 样本:99, alpha优化次数:0
迭代次数: 6
支持向量个数:22
训练集错误率: 1.00%
测试集错误率: 4.00%
[Finished in 1.7s]

```

可以看到，训练集错误率为1%，测试集错误率都是4%，训练耗时1.7s。可以尝试更换不同的K1参数以观察测试错误率、训练错误率、支持向量个数。你会发现K1过大，会出现过拟合的情况，即训练集错误率低，但是测试集错误率高。

五、Sklearn构建SVM分类器

在第一篇文章中，我们使用了kNN进行手写数字识别。它的缺点是存储空间大，因为要保留所有的训练样本，如果你的老板让你节约这个内存空间，识别效果，甚至更好。那这个时候，我们就要可以使用SVM了，因为它只需要保留支持向量即可，而且能获得可比的效果。

使用的数据集还是kNN用到的数据集（testDigits和trainingDigits）：[点我查看](#)

如果对这个数据集不了解的，可以先看看我的第一篇文章：

CSDN：[点我查看](#)

知乎：[点我查看](#)

首先，我们先使用自己用python写的代码进行训练。创建文件svm-digits.py文件，编写代码如下：

```
1 # -*-coding:utf-8 -*-
2 import matplotlib.pyplot as plt
3 import numpy as np
4 import random
5
6 """
7 Author:
8     Jack Cui
9 Blog:
10     http://blog.csdn.net/c406495762
11 Zhihu:
12     https://www.zhihu.com/people/Jack--Cui/
13 Modify:
14     2017-10-03
15 """
16
17 class optStruct:
18     """
19     数据结构，维护所有需要操作的值
20     Parameters:
21         dataMatIn - 数据矩阵
22         classLabels - 数据标签
23         C - 松弛变量
24         toler - 容错率
25         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
26     """
27     def __init__(self, dataMatIn, classLabels, C, toler, kTup):
28         self.X = dataMatIn #数据矩阵
29         self.labelMat = classLabels #数据标签
30         self.C = C #松弛变量
31         self.tol = toler #容错率
32         self.m = np.shape(dataMatIn)[0] #数据矩阵行数
33         self.alphas = np.mat(np.zeros((self.m,1))) #根据矩阵行数初始化alpha参数为0
34         self.b = 0 #初始化b参数为0
35         self.eCache = np.mat(np.zeros((self.m,2))) #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。
36         self.K = np.mat(np.zeros((self.m,self.m))) #初始化核K
37         for i in range(self.m): #计算所有数据的核K
38             self.K[:,i] = kernelTrans(self.X, self.X[i,:], kTup)
39
40 def kernelTrans(X, A, kTup):
41     """
42     通过核函数将数据转换更高维的空间
43     Parameters:
44         X - 数据矩阵
45         A - 单个数据的向量
46         kTup - 包含核函数信息的元组
47     Returns:
48         K - 计算的核K
49     """
50     m,n = np.shape(X)
51     K = np.mat(np.zeros((m,1)))
52     if kTup[0] == 'lin': K = X * A.T #线性核函数,只进行内积。
53     elif kTup[0] == 'rbf': #高斯核函数,根据高斯核函数公式进行计算
54         for j in range(m):
55             deltaRow = X[j,:] - A
56             K[j] = deltaRow*deltaRow.T
57         K = np.exp(K/(-1*kTup[1]**2)) #计算高斯核K
58     else: raise NameError('核函数无法识别')
59     return K #返回计算的核K
60
61 def loadDataSet(fileName):
62     """
63     读取数据
64     Parameters:
65         fileName - 文件名
66     Returns:
67         dataMat - 数据矩阵
68         labelMat - 数据标签
69     """
70     dataMat = []; labelMat = []
71     fr = open(fileName)
72     for line in fr.readlines(): #逐行读取，滤除空格等
73         lineArr = line.strip().split('\t')
74         dataMat.append([float(lineArr[0]), float(lineArr[1])]) #添加数据
75         labelMat.append(float(lineArr[2])) #添加标签
76     return dataMat,labelMat
77
78 def calcEk(oS, k):
```

```

79     """
80     计算误差
81     Parameters:
82         oS - 数据结构
83         k - 标号为k的数据
84     Returns:
85         Ek - 标号为k的数据误差
86     """
87     fXk = float(np.multiply(oS.alphas,oS.labelMat).T*oS.K[:,k] + oS.b)
88     Ek = fXk - float(oS.labelMat[k])
89     return Ek
90
91 def selectJrand(i, m):
92     """
93     函数说明:随机选择alpha_j的索引值
94
95     Parameters:
96         i - alpha_i的索引值
97         m - alpha参数个数
98     Returns:
99         j - alpha_j的索引值
100    """
101    j = i                                # 选择一个不等于i的j
102    while (j == i):
103        j = int(random.uniform(0, m))
104    return j
105
106 def selectJ(i, oS, Ei):
107     """
108     内循环启发方式2
109     Parameters:
110         i - 标号为i的数据的索引值
111         oS - 数据结构
112         Ei - 标号为i的数据误差
113     Returns:
114         j, maxK - 标号为j或maxK的数据的索引值
115         Ej - 标号为j的数据误差
116     """
117     maxK = -1; maxDeltaE = 0; Ej = 0                                # 初始化
118     oS.eCache[i] = [1,Ei]                                          # 根据Ei更新误差缓存
119     validEcacheList = np.nonzero(oS.eCache[:,0].A)[0]              # 返回误差不为0的数据的索引值
120     if (len(validEcacheList)) > 1:                                   # 有不为0的误差
121         for k in validEcacheList:                                    # 遍历,找到最大的Ek
122             if k == i: continue                                     # 不计算i,浪费时间
123             Ek = calcEk(oS, k)                                       # 计算Ek
124             deltaE = abs(Ei - Ek)                                    # 计算|Ei-Ek|
125             if (deltaE > maxDeltaE):                                  # 找到maxDeltaE
126                 maxK = k; maxDeltaE = deltaE; Ej = Ek
127             return maxK, Ej                                          # 返回maxK,Ej
128     else:                                                            # 没有不为0的误差
129         j = selectJrand(i, oS.m)                                    # 随机选择alpha_j的索引值
130         Ej = calcEk(oS, j)                                           # 计算Ej
131     return j, Ej                                                    # j,Ej
132
133 def updateEk(oS, k):
134     """
135     计算Ek,并更新误差缓存
136     Parameters:
137         oS - 数据结构
138         k - 标号为k的数据的索引值
139     Returns:
140         无
141     """
142     Ek = calcEk(oS, k)                                              # 计算Ek
143     oS.eCache[k] = [1,Ek]                                          # 更新误差缓存
144
145
146 def clipAlpha(aj,H,L):
147     """
148     修剪alpha_j
149     Parameters:
150         aj - alpha_j的值
151         H - alpha上限
152         L - alpha下限
153     Returns:
154         aj - 修剪后的alpah_j的值
155     """
156     if aj > H:
157         aj = H
158     if L > aj:
159         aj = L
160     return aj
161
162 def innerL(i, oS):
163     """
164     优化的SMO算法
165     Parameters:
166         i - 标号为i的数据的索引值
167         oS - 数据结构
168     Returns:
169         1 - 有任意一对alpha值发生变化
170         0 - 没有任意一对alpha值发生变化或变化太小
171     """
172     #步骤1: 计算误差Ei
173     Ei = calcEk(oS, i)
174     #优化alpha,设定一定的容错率。
175     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)):
176         #使用内循环启发方式2选择alpha_j,并计算Ej
177         j,Ej = selectJ(i, oS, Ei)
178         #保存更新前的alpha值,使用深拷贝
179         alphaIold = oS.alphas[i].copy(); alphaJold = oS.alphas[j].copy();
180         #步骤2: 计算上下界L和H
181         if (oS.labelMat[i] != oS.labelMat[j]):
182             L = max(0, oS.alphas[j] - oS.alphas[i])

```



```

183     H = min(oS.C, oS.C + oS.alphas[j] - oS.alphas[i])
184     else:
185         L = max(0, oS.alphas[j] + oS.alphas[i] - oS.C)
186         H = min(oS.C, oS.alphas[j] + oS.alphas[i])
187     if L == H:
188         print("L==H")
189         return 0
190     #步骤3: 计算eta
191     eta = 2.0 * oS.K[i,j] - oS.K[i,i] - oS.K[j,j]
192     if eta >= 0:
193         print("eta>=0")
194         return 0
195     #步骤4: 更新alpha_j
196     oS.alphas[j] -= oS.labelMat[j] * (Ei - Ej)/eta
197     #步骤5: 修剪alpha_j
198     oS.alphas[j] = clipAlpha(oS.alphas[j],H,L)
199     #更新Ej至误差缓存
200     updateEk(oS, j)
201     if (abs(oS.alphas[j] - alphaJold) < 0.00001):
202         print("alpha_j变化太小")
203         return 0
204     #步骤6: 更新alpha_i
205     oS.alphas[i] += oS.labelMat[j]*oS.labelMat[i]*(alphaJold - oS.alphas[j])
206     #更新Ei至误差缓存
207     updateEk(oS, i)
208     #步骤7: 更新b_1和b_2
209     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]
210     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]
211     #步骤8: 根据b_1和b_2更新b
212     if (0 < oS.alphas[i]) and (oS.C > oS.alphas[i]): oS.b = b1
213     elif (0 < oS.alphas[j]) and (oS.C > oS.alphas[j]): oS.b = b2
214     else: oS.b = (b1 + b2)/2.0
215     return 1
216 else:
217     return 0
218
219 def smoP(dataMatIn, classLabels, C, toler, maxIter, kTup = ('lin',0)):
220     """
221     完整的线性SMO算法
222     Parameters:
223         dataMatIn - 数据矩阵
224         classLabels - 数据标签
225         C - 松弛变量
226         toler - 容错率
227         maxIter - 最大迭代次数
228         kTup - 包含核函数信息的元组
229     Returns:
230         oS.b - SMO算法计算的b
231         oS.alphas - SMO算法计算的alphas
232     """
233     oS = optStruct(np.mat(dataMatIn), np.mat(classLabels).transpose(), C, toler, kTup)
234     iter = 0
235     entireSet = True; alphaPairsChanged = 0
236     while (iter < maxIter) and ((alphaPairsChanged > 0) or (entireSet)):
237         alphaPairsChanged = 0
238         if entireSet:
239             for i in range(oS.m):
240                 alphaPairsChanged += innerL(i,oS)
241                 print("全样本遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (iter,i,alphaPairsChanged))
242             iter += 1
243         else:
244             nonBoundIs = np.nonzero((oS.alphas.A > 0) * (oS.alphas.A < C))[0]
245             for i in nonBoundIs:
246                 alphaPairsChanged += innerL(i,oS)
247                 print("非边界遍历:第%d次迭代 样本:%d, alpha优化次数:%d" % (iter,i,alphaPairsChanged))
248             iter += 1
249         if entireSet:
250             entireSet = False
251         elif (alphaPairsChanged == 0):
252             entireSet = True
253         print("迭代次数: %d" % iter)
254     return oS.b,oS.alphas
255
256
257 def img2vector(filename):
258     """
259     将32x32的二进制图像转换为1x1024向量。
260     Parameters:
261         filename - 文件名
262     Returns:
263         returnVect - 返回的二进制图像的1x1024向量
264     """
265     returnVect = np.zeros((1,1024))
266     fr = open(filename)
267     for i in range(32):
268         lineStr = fr.readline()
269         for j in range(32):
270             returnVect[0,32*i+j] = int(lineStr[j])
271     return returnVect
272
273 def loadImages(dirName):
274     """
275     加载图片
276     Parameters:
277         dirName - 文件夹的名字
278     Returns:
279         trainingMat - 数据矩阵
280         hwLabels - 数据标签
281     """
282     from os import listdir
283     hwLabels = []
284     trainingFileList = listdir(dirName)
285     m = len(trainingFileList)
286     trainingMat = np.zeros((m,1024))

```

```

#初始化数据结构
#初始化当前迭代次数
#遍历整个数据集都alpha也没有更新或者超过最大迭代次数
#遍历整个数据集
#使用优化的SMO算法
#遍历非边界值
#遍历不在边界0和C的alpha
#遍历一次后改为非边界遍历
#如果alpha没有更新,计算全样本遍历
#返回SMO算法计算的b和alphas

```

```

287     for i in range(m):
288         fileNameStr = trainingFileList[i]
289         fileStr = fileNameStr.split('.')[0]
290         classNumStr = int(fileStr.split('_')[0])
291         if classNumStr == 9: hwLabels.append(-1)
292         else: hwLabels.append(1)
293         trainingMat[i,:] = img2vector('%s/%s' % (dirName, fileNameStr))
294     return trainingMat, hwLabels
295
296 def testDigits(kTup=('rbf', 10)):
297     """
298     测试函数
299     Parameters:
300         kTup - 包含核函数信息的元组
301     Returns:
302         无
303     """
304     dataArr, labelArr = loadImages('trainingDigits')
305     b, alphas = smop(dataArr, labelArr, 200, 0.0001, 10, kTup)
306     datMat = np.mat(dataArr); labelMat = np.mat(labelArr).transpose()
307     svInd = np.nonzero(alphas.A>0)[0]
308     sVs=datMat[svInd]
309     labelSV = labelMat[svInd];
310     print("支持向量个数:%d" % np.shape(sVs)[0])
311     m,n = np.shape(datMat)
312     errorCount = 0
313     for i in range(m):
314         kernelEval = kernelTrans(sVs,datMat[i,:],kTup)
315         predict=kernelEval.T * np.multiply(labelSV,alphas[svInd]) + b
316         if np.sign(predict) != np.sign(labelArr[i]): errorCount += 1
317     print("训练集错误率: %.2f%%" % (float(errorCount)/m))
318     dataArr, labelArr = loadImages('testDigits')
319     errorCount = 0
320     datMat = np.mat(dataArr); labelMat = np.mat(labelArr).transpose()
321     m,n = np.shape(datMat)
322     for i in range(m):
323         kernelEval = kernelTrans(sVs,datMat[i,:],kTup)
324         predict=kernelEval.T * np.multiply(labelSV,alphas[svInd]) + b
325         if np.sign(predict) != np.sign(labelArr[i]): errorCount += 1
326     print("测试集错误率: %.2f%%" % (float(errorCount)/m))
327
328 if __name__ == '__main__':
329     testDigits()

```

SMO算法实现部分跟上文是一样的，我们新建了img2vector()、loadImages()、testDigits()函数，它们分别用于二进制图形转换、图片加载、训练。我们自己的SVM分类器是个二类分类器，所以在设置标签的时候，将9作为负类，其余的0-8作为正类，进行训练。这是一种'ovr'思想，即one vs rest，就和剩余所有的类别进行分类。如果想实现10个数字的识别，一个简单的方法是，训练出10个分类器。这里简单起见，只训练了一个用于分类9和其余所有数字的分类器，运行结果如下：

```

328 if __name__ == '__main__':
329     testDigits()

```

```

工作本遍历:第3次迭代 样本:1922, alpha优化次数:0
alpha_j变化太小
全样本遍历:第3次迭代 样本:1923, alpha优化次数:0
alpha_j变化太小
全样本遍历:第3次迭代 样本:1924, alpha优化次数:0
全样本遍历:第3次迭代 样本:1925, alpha优化次数:0
alpha_j变化太小
全样本遍历:第3次迭代 样本:1926, alpha优化次数:0
alpha_j变化太小
全样本遍历:第3次迭代 样本:1927, alpha优化次数:0
alpha_j变化太小
全样本遍历:第3次迭代 样本:1928, alpha优化次数:0
alpha_j变化太小
全样本遍历:第3次迭代 样本:1929, alpha优化次数:0
alpha_j变化太小
全样本遍历:第3次迭代 样本:1930, alpha优化次数:0
alpha_j变化太小
全样本遍历:第3次迭代 样本:1931, alpha优化次数:0
全样本遍历:第3次迭代 样本:1932, alpha优化次数:0
alpha_j变化太小
全样本遍历:第3次迭代 样本:1933, alpha优化次数:0
迭代次数: 4
支持向量个数:298
训练集错误率: 0.00%
测试集错误率: 0.01%
[Finished in 307.4s]

```

可以看到，虽然我们进行了所谓的“优化”，但是训练仍然很耗时，迭代10次，花费了307.4s。因为我们没有多进程、没有设置自动的终止条件，总之优化的地方太多了。尽管如此，我们训练后得到的结果还是不错的，可以看到训练集错误率为0，测试集错误率也仅为0.01%。

接下来，就是讲解本文的重头戏：sklearn.svm.SVC。

1、sklearn.svm.SVC

官方英文文档手册：[点我查看](#)

sklearn.svm模块提供了很多模型供我们使用，本文使用的是svm.SVC，它是基于libsvm实现的。

sklearn.svm: Support Vector Machines

The `sklearn.svm` module includes Support Vector Machine algorithms.

User guide: See the [Support Vector Machines](#) section for further details.

Estimators

<code>svm.LinearSVC</code> ([penalty, loss, dual, tol, C, ...])	Linear Support Vector Classification.
<code>svm.LinearSVR</code> ([epsilon, tol, C, loss, ...])	Linear Support Vector Regression.
<code>svm.NuSVC</code> ([nu, kernel, degree, gamma, ...])	Nu-Support Vector Classification.
<code>svm.NuSVR</code> ([nu, C, kernel, degree, gamma, ...])	Nu Support Vector Regression.
<code>svm.OneClassSVM</code> ([kernel, degree, gamma, ...])	Unsupervised Outlier Detection.
<code>svm.SVC</code> ([C, kernel, degree, gamma, coef0, ...])	C-Support Vector Classification.
<code>svm.SVR</code> ([kernel, degree, gamma, coef0, tol, ...])	Epsilon-Support Vector Regression.
<code>svm.l1_min_c</code> (X, y[, loss, fit_intercept, ...])	Return the lowest bound for C such that for C in (l1_min_C, infinity) the model is guaranteed not to be empty.

Low-level methods

<code>svm.libsvm.cross_validation</code>	Binding of the cross-validation routine (low-level routine)
<code>svm.libsvm.decision_function</code>	Predict margin (libsvm name for this is <code>predict_values</code>)
<code>svm.libsvm.fit</code>	Train the model using libsvm (low-level method)
<code>svm.libsvm.predict</code>	Predict target values of X given a model (low-level method)
<code>svm.libsvm.predict_proba</code>	Predict probabilities

让我们先看下SVC这个函数，一共有14个参数：

sklearn.svm.SVC

```
class sklearn.svm.SVC(C=1.0, kernel='rbf', degree=3, gamma='auto', coef0=0.0, shrinking=True, probability=False,
tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr',
random_state=None)
```

[\[source\]](#)

参数说明如下：

- **C**：惩罚项，float类型，可选参数，默认为1.0，C越大，即对分错样本的惩罚程度越大，因此在训练样本中准确率越高，但是泛化能力降低，也就数据的分类准确率降低。相反，减小C的话，容许训练样本中有一些误分类错误样本，泛化能力强。对于训练样本带有噪声的情况，一般采用后者本集中错误分类的样本作为噪声。
- **kernel**：核函数类型，str类型，默认为'rbf'。可选参数为：
 - 'linear'：线性核函数
 - 'poly'：多项式核函数
 - 'rbf'：径向核函数/高斯核
 - 'sigmoid'：sigmoid核函数
 - 'precomputed'：核矩阵
 precomputed表示自己提前计算好核函数矩阵，这时候算法内部就不再用核函数去计算核矩阵，而是直接用你给的核矩阵，核矩阵需要为n的。
- **degree**：多项式核函数的阶数，int类型，可选参数，默认为3。这个参数只对多项式核函数有用，是指多项式核函数的阶数n，如果给的核函数参数核函数，则会忽略该参数。
- **gamma**：核函数系数，float类型，可选参数，默认为auto。只对'rbf'、'poly'、'sigmoid'有效。如果gamma为auto，代表其值为样本特征数的倒数1/n_features。
- **coef0**：核函数中的独立项，float类型，可选参数，默认为0.0。只有对'poly'和'sigmoid'核函数有用，是指其中的参数c。
- **probability**：是否启用概率估计，bool类型，可选参数，默认为False，这必须在调用fit()之前启用，并且会fit()方法速度变慢。
- **shrinking**：是否采用启发式收缩方式，bool类型，可选参数，默认为True。
- **tol**：svm停止训练的误差精度，float类型，可选参数，默认为1e^-3。
- **cache_size**：内存大小，float类型，可选参数，默认为200。指定训练所需要的内存，以MB为单位，默认为200MB。
- **class_weight**：类别权重，dict类型或str类型，可选参数，默认为None。给每个类别分别设置不同的惩罚参数C，如果没有给，则会给所有类别都即前面参数指出的参数C。如果给定参数'balance'，则使用y的值自动调整与输入数据中的类频率成反比的权重。
- **verbose**：是否启用详细输出，bool类型，默认为False，此设置利用libsvm中的每个进程运行时设置，如果启用，可能无法在多线程上下文中正常工作，一般情况都设为False，不用管它。

- **max_iter**：最大迭代次数，int类型，默认为-1，表示不限制。
- **decision_function_shape**：决策函数类型，可选参数'ovo'和'ovr'，默认为'ovr'。'ovo'表示one vs one，'ovr'表示one vs rest。
- **random_state**：数据洗牌时的种子值，int类型，可选参数，默认为None。伪随机数发生器的种子,在混洗数据时用于概率估计。

其实，只要自己写了SMO算法，每个参数的意思，大概都是能明白的。

2、编写代码

SVC很是强大，我们不用理解算法实现的具体细节，不用理解算法的优化方法。同时，它也满足我们的多分类需求。创建文件svm-svc.py文件，编写

```

1  #-*- coding: UTF-8 -*-
2  import numpy as np
3  import operator
4  from os import listdir
5  from sklearn.svm import SVC
6
7  """
8  Author:
9      Jack Cui
10 Blog:
11     http://blog.csdn.net/c406495762
12 Zhihu:
13     https://www.zhihu.com/people/Jack--Cui/
14 Modify:
15     2017-10-04
16 """
17
18 def img2vector(filename):
19     """
20     将32x32的二进制图像转换为1x1024向量。
21     Parameters:
22         filename - 文件名
23     Returns:
24         returnVect - 返回的二进制图像的1x1024向量
25     """
26     #创建1x1024零向量
27     returnVect = np.zeros((1, 1024))
28     #打开文件
29     fr = open(filename)
30     #按行读取
31     for i in range(32):
32         #读一行数据
33         lineStr = fr.readline()
34         #每一行的前32个元素依次添加到returnVect中
35         for j in range(32):
36             returnVect[0, 32*i+j] = int(lineStr[j])
37     #返回转换后的1x1024向量
38     return returnVect
39
40 def handwritingClassTest():
41     """
42     手写数字分类测试
43     Parameters:
44         无
45     Returns:
46         无
47     """
48     #测试集的Labels
49     hwLabels = []
50     #返回trainingDigits目录下的文件名
51     trainingFileList = listdir('trainingDigits')
52     #返回文件夹下文件的个数
53     m = len(trainingFileList)
54     #初始化训练的Mat矩阵,测试集
55     trainingMat = np.zeros((m, 1024))
56     #从文件名中解析出训练集的类别
57     for i in range(m):
58         #获得文件的名字
59         fileNameStr = trainingFileList[i]
60         #获得分类的数字
61         classNumber = int(fileNameStr.split('.')[0])
62         #将获得的类别添加到hwLabels中
63         hwLabels.append(classNumber)
64         #将每一个文件的1x1024数据存储在trainingMat矩阵中
65         trainingMat[i, :] = img2vector('trainingDigits/%s' % (fileNameStr))
66     clf = SVC(C=200, kernel='rbf')
67     clf.fit(trainingMat, hwLabels)
68     #返回testDigits目录下的文件列表
69     testFileList = listdir('testDigits')
70     #错误检测计数
71     errorCount = 0.0
72     #测试数据的数量
73     mTest = len(testFileList)
74     #从文件中解析出测试集的类别并进行分类测试
75     for i in range(mTest):
76         #获得文件的名字
77         fileNameStr = testFileList[i]
78         #获得分类的数字
79         classNumber = int(fileNameStr.split('.')[0])
80         #获得测试集的1x1024向量,用于训练
81         vectorUnderTest = img2vector('testDigits/%s' % (fileNameStr))
82         #获得预测结果
83         # classifierResult = classify0(vectorUnderTest, trainingMat, hwLabels, 3)
84         classifierResult = clf.predict(vectorUnderTest)
85         print("分类返回结果为%d\t真实结果为%d" % (classifierResult, classNumber))
86         if(classifierResult != classNumber):
87             errorCount += 1.0
88     print("总共错了%d个数据\n错误率为%f%%" % (errorCount, errorCount/mTest * 100))
89

```

