

## 6.5 利用完整 Platt SMO 算法加速优化

### 6.5.1 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. 刚才已经讨论了简化版SMO算法, 下面我们就讨论完整版的Platt SMO算法. 在这两个版本中, 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. 或许读者已经意识到, 上一节的例子在执行时存在一定的时间提升空间.

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中实现单用圆圈标记的支持向量100 第 6 章 支持向量机遍扫描. 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 $E_j$ . 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 $E_i - E_j$ 最大的alpha值.

### 6.5.2 完整代码

```
import numpy as np
import random
import matplotlib.pyplot as plt
"""
类说明: 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
"""

class optStruct:
    def __init__(self, dataMatIn, classLabels, C, toler):
        self.X = dataMatIn
        self.labelMat = classLabels
        self.C = C
        self.tol = toler
        self.m = np.shape(dataMatIn)[0]
        self.alphas = np.mat(np.zeros((self.m, 1)))
        self.b = 0
        #误差缓存
        self.eCache = np.mat(np.zeros((self.m, 2)))

"""
函数说明: 读取数据
Parameters:
```

```

        fileName - 文件名
Returns:
    dataMat - 数据矩阵(list类型)
    labelMat - 数据标签(list类型)
"""

def loadDataSet(fileName):
    #初始化返回值
    dataMat = []; labelMat = []
    with open(fileName) as f:
        for line in f.readlines():
            lineArr = line.strip().split('\t')
            dataMat.append([float(lineArr[0]), float(lineArr[1])])
            labelMat.append(float(lineArr[-1]))
    return dataMat, labelMat

"""
函数说明：计算误差
Parameters:
    oS - 数据结构
    k - 标号为k的数据
Returns:
    Ek - 标号为k的数据的误差
"""
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

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

"""
函数说明：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):
    #初始化
    maxK = -1; maxDeltaE = 0; Ej = 0
    #根据Ei更新误差缓存
    oS.eCache[i] = [1, Ei]
    #返回误差不为0的数据的索引值
    validEcacheList = np.nonzero(oS.eCache[:,0].A)[0]
    if (len(validEcacheList)) > 1:
        for k in validEcacheList:
            if k == i:
                continue
            Ek = calcEk(oS, k)
            deltaE = abs(Ei - Ek)
            if (deltaE > maxDeltaE):
                maxK = k
                maxDeltaE = deltaE
                Ej = Ek
        return maxK, Ej
    else:
        j = selectJrand(i, oS.m)
        Ej = calcEk(oS, j)
    return j, Ej

```

"""

函数说明：计算 $E_k$ ，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)
    oS.eCache[k] = [1, Ek]

```

"""

函数说明：修剪 $\alpha$

Parameters:

aj -  $\alpha$ 值

H -  $\alpha$ 上界

L -  $\alpha$ 下界

Returns:

aj -  $\alpha$ 值

"""

```

def clipAlpha(aj, H, L):
    if aj > H:
        aj = H
    if aj < L:
        aj = L
    return aj

```

"""

函数说明：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.toI) and (oS.alphas[i] < oS.C)) or
    ((oS.labelMat[i] * Ei > oS.toI) 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alpha值, 使用深拷贝
        alphaIoId = oS.alphas[i].copy(); alphaJoId = 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] - alphaJoId) < 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]*(alphaJoId - 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] -
        alphaIoId)*oS.X[i,:]*oS.X[i,:].T - oS.labelMat[j]*(oS.alphas[j] -
        alphaJoId)*oS.X[i,:]*oS.X[j,:].T
        b2 = oS.b - Ej - oS.labelMat[i]*(oS.alphas[i] -
        alphaIoId)*oS.X[i,:]*oS.X[j,:].T - oS.labelMat[j]*(oS.alphas[j] -
        alphaJoId)*oS.X[j,:]*oS.X[j,:].T
        #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

```

```

"""
函数说明：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
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):
    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
    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,则退出循环
        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]
            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

"""
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 showClassifier(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()

def calcws(alphas, dataArr, classLabels):
    """
    计算w
    Parameters:
        dataArr - 数据矩阵
        classLabels - 数据标签
        alphas - alphas值
    Returns:
        w - 计算得到的w
    """
    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

def calcws(alphas, dataArr, classLabels):

```

```

"""
计算w
Parameters:
    dataArr - 数据矩阵
    classLabels - 数据标签
    alphas - alphas值
Returns:
    w - 计算得到的w
"""
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)
    showClassifier(dataArr, classLabels, w, b)

```

运行结果如下:

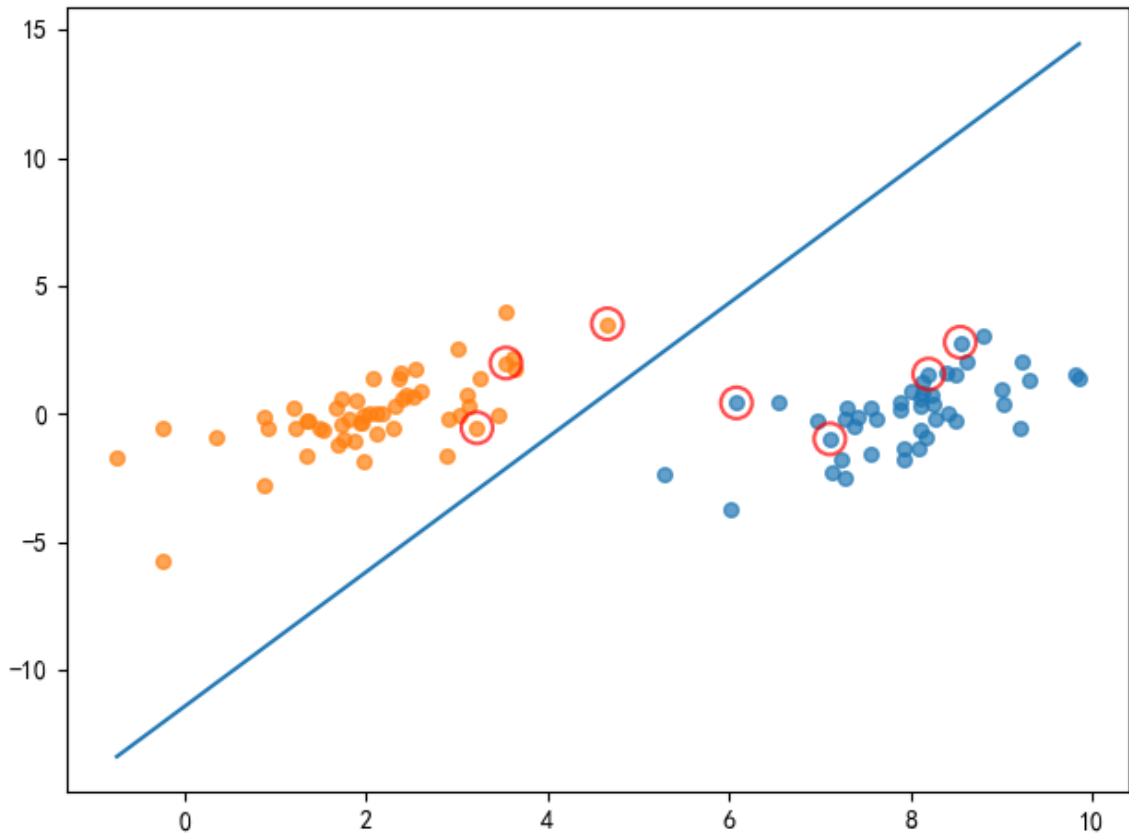
```

svm_Hsu02 x
主样本遍历:第3次迭代 样本:88, alpha优化次数:0
全样本遍历:第3次迭代 样本:89, alpha优化次数:0
全样本遍历:第3次迭代 样本:90, alpha优化次数:0
全样本遍历:第3次迭代 样本:91, alpha优化次数:0
全样本遍历:第3次迭代 样本:92, alpha优化次数:0
全样本遍历:第3次迭代 样本:93, alpha优化次数:0
alpha_j变化太小
全样本遍历:第3次迭代 样本:94, alpha优化次数:0
全样本遍历:第3次迭代 样本:95, alpha优化次数:0
全样本遍历:第3次迭代 样本:96, alpha优化次数:0
alpha_j变化太小
全样本遍历:第3次迭代 样本:97, alpha优化次数:0
全样本遍历:第3次迭代 样本:98, alpha优化次数:0
全样本遍历:第3次迭代 样本:99, alpha优化次数:0
迭代次数: 4

Process finished with exit code 0

```

画出的图像如下:



## 6.6 sklearn构建SVM分类器

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 predict_values)
<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



## 6.6.1 sklearn.svm.SVC

---

看下SVC这个函数, 里面涉及到14个参数, 具体如下:

# sklearn.svm.SVC

```
class sklearn.svm.SVC (C=1.0, kernel='rbf', degree=3, gamma='auto_deprecated', 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)
```

[\[source\]](#)

C-Support Vector Classification.

The implementation is based on libsvm. The fit time scales at least quadratically with the number of samples and may be impractical beyond tens of thousands of samples. For large datasets consider using `sklearn.linear_model.LinearSVC` or `sklearn.linear_model.SGDClassifier` instead, possibly after a `sklearn.kernel_approximation.Nystroem` transformer.

The multiclass support is handled according to a one-vs-one scheme.

For details on the precise mathematical formulation of the provided kernel functions and how `gamma`, `coef0` and `degree` affect each other, see the corresponding section in the narrative documentation: [Kernel functions](#).

Read more in the [User Guide](#).

**Parameters:** **C : float, optional (default=1.0)**

Penalty parameter C of the error term.

**kernel : string, optional (default='rbf')**

Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape `(n_samples, n_samples)`.

**degree : int, optional (default=3)**

Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

**gamma : float, optional (default='auto')**

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

Current default is 'auto' which uses  $1 / n_{\text{features}}$ , if `gamma='scale'` is passed then it uses  $1 / (n_{\text{features}} * X.\text{var}())$  as value of gamma. The current default of gamma, 'auto', will change to 'scale' in version 0.22. 'auto\_deprecated', a deprecated version of 'auto' is used as a default indicating that no explicit value of gamma was passed.

**coef0 : float, optional (default=0.0)**

Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

**shrinking : boolean, optional (default=True)**

Whether to use the shrinking heuristic.

**probability : boolean, optional (default=False)**

Whether to enable probability estimates. This must be enabled prior to calling `fit`, and will slow down that method.

**tol : float, optional (default=1e-3)**

Tolerance for stopping criterion.

**cache\_size : float, optional**

Specify the size of the kernel cache (in MB).

**class\_weight : {dict, 'balanced'}, optional**

Set the parameter C of class `i` to `class_weight[i]*C` for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of `y` to automatically adjust weights inversely proportional to class frequencies in the input data as  $n_{\text{samples}} / (n_{\text{classes}} * \text{np.bincount}(y))$

**verbose : bool, default: False**

Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

**max\_iter : int, optional (default=-1)**

Hard limit on iterations within solver, or -1 for no limit.

**decision\_function\_shape : 'ovo', 'ovr', default='ovr'**

Whether to return a one-vs-rest ('ovr') decision function of shape `(n_samples, n_classes)` as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape `(n_samples, n_classes * (n_classes - 1) / 2)`. However, one-vs-one ('ovo') is always used as multi-class strategy

multi-class strategy.

Changed in version 0.19: `decision_function_shape` is 'ovr' by default.

New in version 0.17: `decision_function_shape='ovr'` is recommended.

Changed in version 0.17: Deprecated `decision_function_shape='ovo'` and `None`.

**random\_state : int, RandomState instance or None, optional (default=None)**

The seed of the pseudo random number generator used when shuffling the data for probability estimates. If int, `random_state` is the seed used by the random number generator; If `RandomState` instance, `random_state` is the random number generator; If `None`, the random number generator is the `RandomState` instance used by `np.random`.

## 参数说明:

各个参数的具体含义如下:

- **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, 也就是对测试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, 把训练样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 $n \times n$ 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是其他核函数, 则会自动忽略该参数.
- **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, 即 $1/n\_features$ .
- **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, 并且会使`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, 则会给所有类别都给 $C=1$ , 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.

## 6.6.2 kNN手写识别回顾

之前关于手写识别的算法, 讲到了kNN, 现在回顾一下用kNN实现的具体的代码, 这里的处理文件处理方式等下会用到

```
import numpy as np
import operator
from os import listdir

"""
函数说明: KNN算法, 分类器

Parameters:
    inx - 用于分类数据(即测试集)
    dataSet - 用于训练的数据(即训练集)
    labels - 分类的标签(即什么类型的电影)
    k - KNN算法参数, 选择距离最小的k个点

Returns:
    sortedClassCount[0][0] - 分类结果
"""

def classify0(inx, dataSet, labels, k):
    #shape[0]返回的是dataSet的函数
    dataSetSize = dataSet.shape[0]
    #np.tile(a,(b,c))函数含义是在列向量方向上重复a共b次(横向), 在行向量方向上重复a共c次(纵向)
    #np.tile(inx, (dataSetSize,1))的含义就是构建dataSetSize行inx,接着减去DataSet对应的.
    diffMat = np.tile(inx, (dataSetSize,1)) - dataSet
    #求得每一行差额的平方
    sqDiffMat = diffMat**2
    #然后求和, axis=1是按行求和,axis=0是按列求和
    sqDistances = sqDiffMat.sum(axis=1)
    #开平方,求得距离
    distances = sqDistances**0.5
    #numpy中的argsort()方法(也是numpy的顶级函数),返回distances中元素从小到大排列后的索引值.等价:a.argsort()==np.argsort(a)
    sortedDistIndices = distances.argsort()
    #构建一个空的字典,用于记录类别次数
    classCount = {}
    for i in range(k):
        #提取出前k个元素的类别
        voteIlabel = labels[sortedDistIndices[i]]
        #字典的get方法,返回指定键的值,如果指定键不存在,那么返回默认值,这里是0.
        #比如键是"爱情片",最开始字典是空的,那么返回0,后面加了1,那么就是{"爱情片":1}
        #接着如果还是"爱情片",那么返回1,再加1,就是2,那么字典变成了{"爱情片":2}
        classCount[voteIlabel] = classCount.get(voteIlabel, 0) + 1
    #sorted函数,里面有参数key,根据一定的方式排序.这里的key=operator.itemgetter(1)表示的是根据字典的值进行排序
    #参数reverse表示是否进行降序排序,True表示是,默认否,即false
    sortedClassCount = sorted(classCount.items(),
                              key=operator.itemgetter(1),reverse=True)
```

```
#则第一个字典的键就是[0][0],也就是返回次数最多的类别,就是我们要的最终类别
return sortedClassCount[0][0]
```

"""

函数说明:将32x32的二进制图像转换称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):
    #创建1x1024零向量
    returnVect = np.zeros((1,1024))
    #打开文件
    with open(filename) as fr:
        #按行读取,注意readlines是读取所有行,readline是读取一行
        for i in range(32):
            #读取每一行数据
            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只有一行,也就是0,i和j都是从0开始,31结束,共32
                returnVect[0,32*i+j] = int(lineStr[j])
    #返回转换后的1x1024向量
    return returnVect
```

"""

函数说明:手写数字分类测试

Parameters:

无

Returns:

无

"""

```
def handwritingClassTest():
    #测试集的Labels
    hwLabels = []
    #返回训练文件夹trainingDigits目录下的文件名,listdir返回的是列表
    trainingFileList = listdir('trainingDigits')
    #返回训练文件夹下文件的个数
    m = len(trainingFileList)
    #初始化训练的矩阵
    trainingMat = np.zeros((m,1024))
    #从文件名中解析出训练集的分类
    for i in range(m):
        #获取文件的名字
        filenameStr = trainingFileList[i]
        #获得分类的数字,因为文件名是'真实数字_文件序列号.txt',所以[0]就是真实数字
        #书上多了一步:fileStr = fileNameStr.split('.')[0],这步是获取数字如:0_3,没有
        classNumStr = int(filenameStr.split('_')[0])
        #将获取的真实数字(也就是真实类别)添加到hwLabels中
        hwLabels.append(classNumStr)
        #将每个文件的1x1024数据存储在trainingMat矩阵中,trainingMat循环完成后,共 mx1024
        trainingMat[i,:] = img2vector('trainingDigits/%s' % (filenameStr))
    #返回测试文件夹testDigits目录下的文件列表
```

```

testFileList = listdir('testDigits')
#测试集数据的数量
mTest = len(testFileList)
#错误检测计数
errorCount = 0
#从文件名中解析出测试集的分类进行分类测试
#这部分的代码类似于上面训练集中的代码
for i in range(mTest):
    #获取文件的名称
    filenameStr = testFileList[i]
    #获得分类的数字,因为文件名是'真实数字_文件序列号.txt',所以[0]就是真实数字
    classNumStr = int(filenameStr.split('_')[0])
    #获取测试集的1x1024向量,就一个1x1024,用于预测最后的数字
    vectorUnderTest = img2vector('testDigits/%s' % (filenameStr))
    #获得预测结果
    classifierResult = classify0(vectorUnderTest, trainingMat, hwLabels, 3)
    print("分类的结果为%d\t\t真实的结果为%d" % (classifierResult, classNumStr))
    if classifierResult != classNumStr:
        errorCount += 1
    print("总共错了%d个数据\n错误率为%f%%" % (errorCount, errorCount /
float(mTest)))

if __name__ == '__main__':
    handwritingClassTest()

```

## 6.6.3 sklearn构建svm分类器--进行手写识别

那利用sklearn构建的svm分类器用于手写识别, 是怎样的,

首先, 关于图像处理函数img2vector是一样的, 而handwritingClassTest函数里, 用sklearn构建的svm代替之前我们自己写的knn.

具体代码如下:

```

import numpy as np
import operator
from os import listdir
from sklearn.svm import SVC

"""
函数说明:将32x32的二进制图像转换称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):
    #创建1x1024零向量
    returnVect = np.zeros((1,1024))
    #打开文件
    with open(filename) as fr:
        #按行读取, 注意readlines是读取所有行, readline是读取一行

```



```

        for i in range(32):
            #读取每一行数据
            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只有一行,也就是0,i和j都是从0开始,31结束,共32
                returnVect[0,32*i+j] = int(lineStr[j])
    #返回转换后的1x1024向量
    return returnVect

"""
函数说明:手写数字分类测试

Parameters:
    无
Returns:
    无
"""
def handwritingClassTest():
    #测试集的Labels
    hwLabels = []
    #返回训练文件夹trainingDigits目录下的文件名, listdir返回的是列表
    trainingFileList = listdir('trainingDigits')
    #返回训练文件夹下文件的个数
    m = len(trainingFileList)
    #初始化训练的矩阵
    trainingMat = np.zeros((m,1024))
    #从文件名中解析出训练集的分类
    for i in range(m):
        #获取文件的名字
        filenameStr = trainingFileList[i]
        #获得分类的数字,因为文件名是'真实数字_文件序列号.txt',所以[0]就是真实数字
        #书上多了一步:fileStr = fileNameStr.split('.')[0], 这步是获取数字如:0_3, 没有
        #必要.
        classNumStr = int(filenameStr.split('_')[0])
        #将获取的真实数字(也就是真实类别)添加到hwLabels中
        hwLabels.append(classNumStr)
        #将每个文件的1x1024数据存储在trainingMat矩阵中,trainingMat循环完成后,共 mx1024
        trainingMat[i,:] = img2vector('trainingDigits/%s' % (filenameStr))

    clf = SVC(C=200, kernel='rbf')
    clf.fit(trainingMat, hwLabels)
    # 返回测试文件夹testDigits目录下的文件列表
    testFileList = listdir('testDigits')
    # 测试集数据的数量
    mTest = len(testFileList)
    # 错误检测计数
    errorCount = 0
    # 从文件名中解析出测试集的分类进行分类测试
    # 这部分的代码类似于上面训练集中的代码
    for i in range(mTest):
        # 获取文件的名字
        filenameStr = testFileList[i]
        # 获得分类的数字,因为文件名是'真实数字_文件序列号.txt',所以[0]就是真实数字
        classNumStr = int(filenameStr.split('_')[0])
        # 获取测试集的1x1024向量,就一个1x1024,用于预测最后的数字
        vectorUnderTest = img2vector('testDigits/%s' % (filenameStr))
        # 获得预测结果

```

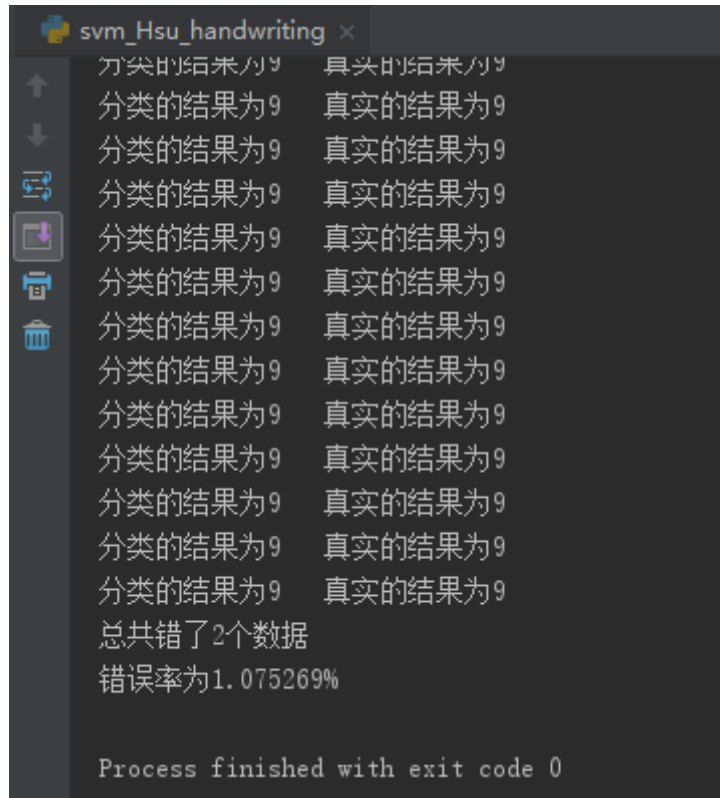
```

classifierResult = clf.predict(vectorUnderTest)
print("分类的结果为%d\t真实的结果为%d" % (classifierResult, classNumStr))
if classifierResult != classNumStr:
    errorCount += 1
print("总共错了%d个数据\n错误率为%f%%" % (errorCount, errorCount /
float(mTest)*100))

if __name__ == '__main__':
    handwritingClassTest()

```

运行结果如下:



```

svm_Hsu_handwriting x
分类的结果为9 真实的结果为9
分类的结果为9 真实的结果为9
分类的结果为9 真实的结果为9
分类的结果为9 真实的结果为9
分类的结果为9 真实的结果为9
分类的结果为9 真实的结果为9
分类的结果为9 真实的结果为9
分类的结果为9 真实的结果为9
分类的结果为9 真实的结果为9
分类的结果为9 真实的结果为9
分类的结果为9 真实的结果为9
分类的结果为9 真实的结果为9
总共错了2个数据
错误率为1.075269%

Process finished with exit code 0

```

整个改变的就是把kNN的分类器-classify0函数, 用sklearn.svm.svc代替了, 其他基本没什么变化.