实验介绍

1. 实验内容

本实验介绍如何使用SVM算法解决手写识别问题，并介绍如何使用sklearn库创建svm模型。

2. 实验目标

通过本实验掌握SVM算法如何应用。

3. 实验知识点

SVM支持向量机原理

4. 实验环境

python 3.6.5

5. 预备知识

概率论与数理统计
Linux命令基本操作
Python编程基础

准备工作

点击屏幕右上方的下载实验数据模块，选择下载svm_hand_digits.tgz到指定目录下，然后再依次选择点击上方的File->Open->Upload,上传刚才下载的数据集压缩包，再使用如下命令解压：

In [1]:
```
!tar -zxvf svm_hand_digits.tgz
```

```
svm_hand_digits/
svm_hand_digits/testDigits/
svm_hand_digits/testDigits/0_0.txt
svm_hand_digits/testDigits/0_1.txt
svm_hand_digits/testDigits/0_10.txt
svm_hand_digits/testDigits/0_11.txt
svm_hand_digits/testDigits/0_12.txt
svm_hand_digits/testDigits/0_13.txt
svm_hand_digits/testDigits/0_14.txt
svm_hand_digits/testDigits/0_15.txt
svm_hand_digits/testDigits/0_16.txt
svm_hand_digits/testDigits/0_17.txt
svm_hand_digits/testDigits/0_18.txt
svm_hand_digits/testDigits/0_19.txt
svm_hand_digits/testDigits/0_2.txt
svm_hand_digits/testDigits/0_20.txt
svm_hand_digits/testDigits/0_21.txt
svm_hand_digits/testDigits/0_22.txt
svm_hand_digits/testDigits/0_23.txt
```

本实验的数据集和kNN手写数字识别的数据集相同，在kNN手写数字识别实验中对数据集有详细的介绍，不了解的同学，可以去参考该实验中对数据集的介绍。

基于SVM的手写数字识别

现在我们开始进行基于SVM的手写数字识别实战。

【实验步骤】读取并处理图片数据

本次实验分为以下几个步骤：
(1)收集数据：提供的文本文件
(2)准备数据：基于二值图像构造向量
(3)分析数据：对图像向量进行目测
(4)训练算法：采用两种不同的核函数，并对径向基函数采用不同的设置来运行SMO算法
(5)测试算法：编写test函数来测试不同的核函数并计算错误率

In [5]:
```python
def img2vector(filename):
    """
    将32x32的二进制图像转换为1x1024向量。

    Args:
        filename: 文件名
    Returns:
        returnVect: 返回的二进制图像的1x1024向量
    """
    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


def loadImages(dirName):
    """
    加载图片

    Args:
        dirName: 文件夹的名字
    Returns:
        trainingMat: 数据矩阵
        hwLabels: 数据标签
    """
    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
```

【练习】构建完整SMO算法

本节我们来构建完整的SMO算法：

In [6]:
```python
import random


class optStruct:
    """
    数据结构，维护所有需要操作的值

    Args:
        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
    """

    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
        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。
        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)


def kernelTrans(X, A, kTup):
    """
    通过核函数将数据转换更高维的空间

    Args:
        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
    """
    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


def loadDataSet(fileName):
    """
    读取数据

    Args:
        fileName: 文件名
    Returns:
        dataMat: 数据矩阵
        labelMat: 数据标签
    """
    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


def calcEk(oS, k):
    """
    计算误差

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


def selectJrand(i, m):
    """
```

```
    函数说明:随机选择alpha_j的索引值

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


def selectJ(i, oS, Ei):
    """
    内循环启发方式2

    Args:
        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
    """
    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
```

```python
def updateEk(oS, k):
    """
    计算Ek,并更新误差缓存

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


def clipAlpha(aj, H, L):
    """
    修剪alpha_j

    Args:
        aj: alpha_j的值
        H: alpha上限
        L: alpha下限
    Returns:
        aj: 修剪后的alpah_j的值
    """
    if aj > H:
        aj = H
    if L > aj:
        aj = L
    return aj


def innerL(i, oS):
    """
    优化的SMO算法

    Args:
        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
    """
    #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


def smoP(dataMatIn, classLabels, C, toler, maxIter, kTup=('lin', 0)):
    """
    完整的线性SMO算法

    Args:
        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
    """
    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
    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]  #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
```

【实验步骤】实现测试函数

本节实现测试函数：

In [8]:

```
def testDigits(kTup=('rbf', 10)):
    """
    测试函数

    Args:
        kTup: 包含核函数信息的元组
    Returns:
        无
    """
    dataArr, labelArr = loadImages('svm_hand_digits/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('svm_hand_digits/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()
```

```
L==H
全样本遍历:第0次迭代 样本:0, alpha优化次数:0
全样本遍历:第0次迭代 样本:1, alpha优化次数:1
全样本遍历:第0次迭代 样本:2, alpha优化次数:2
全样本遍历:第0次迭代 样本:3, alpha优化次数:3
```

```
全样本遍历:第0次迭代 样本:4, alpha优化次数:4
全样本遍历:第0次迭代 样本:5, alpha优化次数:5
L==H
全样本遍历:第0次迭代 样本:6, alpha优化次数:5
全样本遍历:第0次迭代 样本:7, alpha优化次数:6
全样本遍历:第0次迭代 样本:8, alpha优化次数:7
全样本遍历:第0次迭代 样本:9, alpha优化次数:8
全样本遍历:第0次迭代 样本:10, alpha优化次数:9
L==H
全样本遍历:第0次迭代 样本:11, alpha优化次数:9
全样本遍历:第0次迭代 样本:12, alpha优化次数:10
全样本遍历:第0次迭代 样本:13, alpha优化次数:11
全样本遍历:第0次迭代 样本:14, alpha优化次数:12
全样本遍历:第0次迭代 样本:15, alpha优化次数:13
```

本代码中，SMO算法实现部分和实验《SVM支持向量机原理》中介绍的算法是一样的，这里新创建了img2vector()、loadImages()、testDigits()函数，它们分别用于二进制图形转换、图片加载、训练SVM分类器。
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，就是对一个类别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。
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4次，花费了大量的时间（5~10分钟左右）。因为我们没有多进程、没有设置自动的终止条件，总之需要优化的地方太多了。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%。
具体有哪些优化方法呢？同学们可以尝试修改kTup参数来查看测试集错误率和训练集错误率，支持向量个数的变化。

【实验步骤】scikit-learn的SVM说明

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

svm.LinearSVC([penalty, loss, dual, tol, C, ...])	Linear Support Vector Classification.
svm.LinearSVR([epsilon, tol, C, loss, ...])	Linear Support Vector Regression.
svm.NuSVC([nu, kernel, degree, gamma, ...])	Nu-Support Vector Classification.
svm.NuSVR([nu, C, kernel, degree, gamma, ...])	Nu Support Vector Regression.
svm.OneClassSVM([kernel, degree, gamma, ...])	Unsupervised Outlier Detection.
svm.SVC([C, kernel, degree, gamma, coef0, ...])	C-Support Vector Classification.
svm.SVR([kernel, degree, gamma, coef0, tol, ...])	Epsilon-Support Vector Regression.
svm.l1_min_c(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

svm.libsvm.cross_validation	Binding of the cross-validation routine (low-level routine)
svm.libsvm.decision_function	Predict margin (libsvm name for this is predict_values)
svm.libsvm.fit	Train the model using libsvm (low-level method)
svm.libsvm.predict	Predict target values of X given a model (low-level method)
svm.libsvm.predict_proba	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'：径像核函数/高斯核
  * 'sigmod'：sigmod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*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' ,'sigmod'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,'sigmod'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。

由于在上一个实验中，我们自己动手实现过SVM，所以每个参数的意思，你应该大概都能明白的。

【练习】基于scikit-learn构建SVM

SVC很强大，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。

In [11]:
```
from os import listdir

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


def img2vector(filename):
    """
    将32x32的二进制图像转换为1x1024向量。

    Args:
        filename: 文件名
    Returns:
        returnVect: 返回的二进制图像的1x1024向量
    """
    #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


def handwritingClassTest():
    """
    手写数字分类测试

    Returns:
        无
    """
    #测试集的Labels
    hwLabels = []
    #返回trainingDigits目录下的文件名
    trainingFileList = listdir('svm_hand_digits/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('svm_hand_digits/trainingDigits/%s' % (fileNameStr))

    clf = SVC(C=300, kernel='rbf', gamma='auto', coef0=0.1)  # 创建SVC对象
    clf.fit(trainingMat, hwLabels)  #调用fit函数进行模型训练

    #返回testDigits目录下的文件列表
    testFileList = listdir('svm_hand_digits/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('svm_hand_digits/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()
```

```
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
```

分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0
分类返回结果为0 真实结果为0

可以看到测试集的错误率仅为1.37%。试着改变SVC的参数，慢慢体会一下吧~

实验总结

通过本实验，您应该能达到以下两个目标：

1. 掌握SVM算法原理。
2. 熟悉SVM算法的初步应用。

参考文献及延伸阅读

参考资料：

- 1.哈林顿，李锐. 机器学习实战 : Machine learning in action[M]. 人民邮电出版社, 2013.
- 2.http://cuijiahua.com/blog/2017/11/ml_8_svm_1.html (http://cuijiahua.com/blog/2017/11/ml_8_svm_1.html)

延伸阅读：

- 1.李航. 统计学习方法[M]. 清华大学出版社, 2012.