MNIST手写体识别实验报告

马佳良 刘群 胡晨琪

地球系统科学研究中心

MNIST 是一个手写体识别的数据库，里面存储了大量的关于手写体的数据，训练数据有60000张图片，测试数据有10000张图片。本文的主要任务是用kNN，Navie Bayes以及SVM的方法进行MNIST手写数字的识别，这3部分工作分别由小组的3 人分工完成，最后将大家工作整合为本次试验报告。小组成员及分工如下表所示：

成员	分工
马佳良	kNN方法的程序实现及参数测试，并编写实验报告相应的部分
刘群	Naive Bayes方法的程序实现及参数测试，并编写实验报告相应的部分
胡晨琪	SVM方法的程序实现及参数测试，并编写实验报告相应的部分

下面分别介绍了采用kNN，Navie Bayes以及SVM方法对MNIST手写体进行识别的过程，每个部分包括了对算法的简单介绍、代码实现过程、实验结果以及各个参数的改变对分类准确率的讨论等。

1 基于KNN算法的手写体识别

1.1 KNN分类算法简介

KNN(K-Nearest Neighbors)分类算法是一种简单有效的分类算法，其核心思想是：如果要确定某一测试样本属于哪一类，就只需找到其在特征空间中的k 个最邻近的样本；这k 个样本中的大多数属于某一个类别，那么就认为该样本也属于这一类。Python可用如下的具体思路实现该算法：

步骤	解释
step1	准备好训练数据以及测试数据，设定参数k
step2	计算测试数据与训练数据的欧氏距离
step3	将距离排序，找到k个距离最近点的标签
step4	统计上述每个标签的个数，个数最多的标签即为测试数据的类别

通过上面流程，我们可以发现KNN算法存在着一些优缺点。优点是简单有效，易于理解实现；不足之处是该算法的准确度还要取决于k的取值，距离的定义等重要参数，也就是说不通的k值、不同的距离算法对于分类的准确度是有很大影响的。但经过试验以及讨论，为了更快速、准确地得到分类结果，下面调用Python的sklearn模块对手写识别进行求解。

1.2 代码实现

```
import numpy as np
from sklearn import neighbors

##### step 1: prepare the train set and test set from mnist database
#train examples: 60000, size: 28*28
train_image = np.fromfile('train-images.idx3-ubyte', dtype=np.uint8)
train_image = (train_image[16:]).reshape([60000,28*28])
```

```
train_label = np.fromfile('train-labels.idx1-ubyte', dtype=np.uint8)
train_label = (train_label[8:])

#test examples: 10000, size: 28*28
test_image = np.fromfile('t10k-images.idx3-ubyte', dtype=np.uint8)
test_image = (test_image[16:]).reshape([10000,28*28])

test_label = np.fromfile('t10k-labels.idx1-ubyte', dtype=np.uint8)
test_label = (test_label[8:])

##### step 2: get the classifier of knn

model = neighbors.KNeighborsClassifier(n_neighbors=1,weights='distance',algorithm='auto',
    leaf_size=30,p=2,metric='euclidean')

##### step 3: train with the traindata and trainlabels

model.fit(train_image,train_label)


##### step 4: make predictions

knn_class = model.predict(test_image)
knnerror = np.sum(knn_class != test_label)*1.0/knn_class.shape[0]

##### print result
print 'The knn classification is',knn_class
print 'The Error rate of knn is',knnerror
```

上述代码可表述为下列的算法：

步骤	解释
step1	准备好训练数据以及测试数据集
step2	取得knn分类器
step3	导入训练数据$train_image$与训练标签$train_label$，进行训练fit
step4	对测试数据$test_image$进行预测分类
step5	计算分类错误率

其中第二步值得特别说明；在取得knn分类器$model = neighbors.KNeighborsClassifier()$时,括号里没有任何设置表示采用默认的算法进行分类。通过查阅，发现我们可以在括号里设置参数（比如k的值，距离的表示方法）来观察k值，距离的不同表示对手写识别的影响，以便找到一个最适合的参数值。下表所列为可修改的参数值：

参数	解释
n_neighbors	k的值，默认5
weights	每个数据点是否有权重，'uniform','distance'
algorithm	计算最近的邻居时用到的算法，'kd_tree','brute','ball_tree',自动找最合适算法'auto'
leaf_size	针对'kd_tree'与'ball_tree'时的leaf size，默认30
metric	选择距离的计算方法，'cityblock','euclidean','canberra'，默认'minkowski'距离
p	当metric为'minkoski'时：1：manhattan距离;2: euclidean距离;p: minkoski距离

1.3 实验结果及讨论

下面是采用默认的neighbors.KNeighborsClassifier()对60000个测试数据测试的结果，错误率为3.12%, 通过ipython测试耗时为9min。

```
The knn classification is [7 2 1 ..., 4 5 6]
The Error rate of knn is 0.0312
CPU times: user 8min 57s, sys: 84.7 ms, total: 8min 57s
Wall time: 9min
```

为了探究是否能够得到准确率更高的手写识别结果，试图对k 值以及距离的算法进行修改。

1.3.1 对距离算法的敏感度

采用manhattan(cityblock)距离时，k=3，错误率为3.6%, 通过ipython 测试耗时为9min 56s:

```
The knn classification is [7 2 1 ..., 4 5 6]
The Error rate of knn is 0.036
CPU times: user 9min 56s, sys: 60 ms, total: 9min 56s
Wall time: 9min 56s
```

采用欧式距离（euclidean）时，k=3，错误率为2.83%, 通过ipython 测试耗时为10min 9s:

```
The knn classification is [7 2 1 ..., 4 5 6]
The Error rate of knn is 0.0283
CPU times: user 10min 9s, sys: 64.8 ms, total: 10min 9s
Wall time: 10min 9s
```

采用canberra距离时，k=3，错误率为3.87%, 通过ipython 测试耗时为15min 10s:

```
The knn classification is [7 2 1 ..., 4 5 6]
The Error rate of knn is 0.0387
CPU times: user 15min 10s, sys: 71.7 ms, total: 15min 11s
Wall time: 15min 10s
```

说明当k值一定，选择欧式距离时，错误率是最小的，因此对于手写识别推荐使用欧氏距离作为计算训练数据与样本数据的距离方法。

1.3.2 对k值的敏感度

下表所列为当我改变k值（距离用的都是欧式距离）时，程序对应的分类错误率与运行时间：

k值	错误率	运行时间
1	3.09%	9min 53s
2	3.09%	11min 10s
3	2.83%	11min 02s
4	2.86%	11min 54s
5	3.09%	11min 39s
6	2.91%	11min 42s
7	3.00%	10min 21s
8	2.94%	10min 46s
9	3.27%	10min 59s
10	3.16%	10min 45s
11	3.22%	13min 16s
12	3.22%	13min 38s
.....		

从该表可以看出ipython运行的时间都在10-13min之间，k值不同对应的错误率也不同，基本上在3%左右。其中当k 为3 时错误率最小为2.83%（后面一些的k 值未列出，从结果看基本上随着k值增大，错误率也最大），也就表明此时的knn 分类算法得到的分类结果最好。

综合上述内容，用knn算法进行手写识别时，错误率最小可达到2.83%（此时k=3,采用欧式距离），但运行时间较长为11min02s。

2 基于Naive Bayes 算法的手写体识别

2.1 Naive Bayes算法简介

Naive Bayes算法是一种基于Bayes概率的一种分类方法，其主要思想是要最大化后验概率。主要假设所有的特征都是相互独立的，所有的特征对结论都是同等重要的。贝叶斯分类的基本公式如下：

$$p(C|X) = \frac{p(X|C)p(C)}{p(X)} \rightarrow p(C|X) \propto p(X|C)p(C)$$

其中，$X = X(X_1, X_2, \cdots, X_n), C = C(c_1, c_2, \cdots, c_L)$. 最终我们要在计算得出的后验概率$p(c_k|X)(k = 1, 2, \cdots, L)$中选择一个最大值，这样$X$就属于这个概率所对应的类别。

2.2 几种贝叶斯实现方法的比较

2.2.1 基于sklearn库的GaussianNB算法

这种方法是假设所有的feature都满足正态分布, 对每一种feature分别进行拟合，利用库实现的时候也比较简单，只需对模型进行训练和预测即可，代码如下(实际运行时请看代码文件mnist_nb_sklearn_GaussianNB.py)：

```
#!/usr/bin/env python
import numpy as np
from sklearn.naive_bayes import GaussianNB

# 导入训练数据和标签, 前面16纪录的是数据的相关信息，故忽略掉bits
train_image = np.fromfile('train-images.idx3-ubyte', dtype=np.uint8)
train_image = (train_image[16:]).reshape([60000,784])

train_label = np.fromfile('train-labels.idx1-ubyte', dtype=np.uint8)
train_label = train_label[8:]

# 导入测试数据和标签
test_image = np.fromfile('t10k-images.idx3-ubyte', dtype=np.uint8)
test_image = (test_image[16:]).reshape([10000,784])

test_label = np.fromfile('t10k-labels.idx1-ubyte', dtype=np.uint8)
test_label = test_label[8:]

# create a Navie Bayes model based on Gaussian
model = GaussianNB()

# train the model
model.fit(train_image, train_label)

# predict
newlabel = model.predict(test_image)
error = sum(test_label != newlabel)*1.0/10000
print "Error rate is " + str(error*100) + "%."
```

对这种方法的测试结果如下,从中可以看出，该方法的错误率高达48.76%, 但是运行时间较短。

```
Error rate is 48.76%.
CPU times: user 942 ms, sys: 448 ms, total: 1.39 s
Wall time: 1.44 s
```

2.2.2 手动实现基于Gauss分布的NB算法(提取某些非零feature)

在这里主要是针对上一种方法的错误率较高，想采取一种新的方法，由于在28×28的像素中，存在着一些全是0的像素点，比如说边界附近的值，这些点在实际计算中是没有什么作用的，因此可以通过一种方法去除这些点的信息，降低feature的维度。我们在这里引入了一个flag变量，其主要用来纪录不同的label所对应的训练数据中，哪些维度全是0，哪些存在着非0 的feature。如果某个feature在所有该label的测试数据中全为0的话，则flag记为false，否则记为true。这样的话，在下面求Gauss分布的均值和方差时，不同的label的数据所对应的维度也不同。还有一点需要指出的是，在实际的计算中，在计算概率的对数值时，有可能出现$log0$的情况，

所以在计算的时候加入了一个小项10^{-100}, 这样就可以顺利的计算下去。下面是部分代码示例，实际代码在文件mnist_nb_less_features.py 中。

```
# 前面导入数据部分同上一个程序，忽略掉
# flag[i] 存放的是为的训练数据中哪些维度全是labeli0
def flagCalculation(train_image, train_label):
    '''
    flag is a dictionary, where the key is the label 0-9, and the values
    are the bool values indicate which column are all zeros
    '''
    flag = {}
    for i in range(10):
        flag[i] = np.array([ train_image[train_label == i, j].sum() != 0 for j in range(784)
    ])
    return flag

flag = flagCalculation(train_image, train_label)

# Calculate the A prioir probablity
p_y = np.array([train_label[train_label == i].shape[0]*1.0 / train_label.shape[0] for i in
    range(10)])

# Calculate the sigma and mu
sigma = {}
mu = {}
# 在计算时只对不全是的进行计算feature0
for i in range(10):
    mu[i]    = np.average(train_image[:, flag[i]][train_label == i, :], axis=0)
    sigma[i] = np.std(train_image[:, flag[i]][train_label == i, :], axis=0)

def calcP_xy(x_test):
    log_p_yx = {}
    for i in range(10):
        x_test_new = x_test[flag[i]]
        # 在计算时防止出现，加上一个很小的数log01e-100
        log_p_yx[i] = np.sum(np.log(1.0/sigma[i]/np.sqrt(2*np.pi)*np.exp(-(x_test_new-mu[i])
    **2/2.0/sigma[i]**2)+1e-100)) + np.log(p_y[i])
    return np.argmax(log_p_yx.values())

newLabel = np.zeros(10000)
for i in range(10000):
    newLabel[i] = calcP_xy(test_image[i,:])

error = sum(test_label != newLabel)*1.0/10000
print "Error rate is " + str(error*100) + "%."
```

对这个程序的测试表明，错误率为35.03%, 比前一种方法提高了十三个百分点, 但是这个方法与KNN方法相比，错误率还是太高，因此下面考虑进一步改进。

```
Error rate is 35.03%.
CPU times: user 10.7 s, sys: 228 ms, total: 11 s
Wall time: 11 s
```

2.2.3 手动实现基于Gauss分布的NB算法(提取某些非零feature并归一化)

这种实现方法与上一种算法的唯一区别是，我们对训练和测试数据进行了归一化处理，即将数据除以它们的最大值255. 下面是部分代码示例，实际代码在文件mnist_nb_less_features_normalized.py 中。

```
train_image = np.fromfile('train-images.idx3-ubyte', dtype=np.uint8)
train_image = (train_image[16:]).reshape([60000,784]) / 255.0

test_image = np.fromfile('t10k-images.idx3-ubyte', dtype=np.uint8)
test_image = (test_image[16:]).reshape([10000,784]) / 255.0
```

我们可以看到，这样做的结果是提高了测试的准确率, 错误率由前一种方法的35.03%, 降到了现在的21.04%.

```
Error rate is 21.04%.
CPU times: user 19.3 s, sys: 2.76 s, total: 22.1 s
Wall time: 22.1 s
```

2.2.4 基于sklearn库中的Multinomial分布

代码方面与基于sklearn库中的GaussianNB方法的唯一区别就是将model语句由

```
model = GaussianNB()
```

改为了

```
model = MultinomialNB(alpha=0.001)
```

(实际代码在文件mnist_nb_sklearn_MultinomialNB.py 中。)

Multinomial Naive Bayes 分类器是应用于离散特征问题的一种算法，比如说文本单词的分类。在这个算法中，需要调节的是一个参数是α, 这个参数是一个平滑参数，用来防止出现概率为0的情况，当$\alpha = 0$时，表示没有平滑，默认值是取1，在这里经过测试之后，我取了$\alpha = 0.001$, 此时，测试结果的错误率为16.3%。同时也可以看出，这种方法的运算速度特别快，运行时间还是在ms级别。

```
Error rate is 16.3%.
CPU times: user 386 ms, sys: 174 ms, total: 560 ms
Wall time: 384 ms
```

2.2.5 基于多元正态分的NB算法

由于对于这些feature的真实分布我们无法获得，因此也就很难得到真实的似然概率，在这里我们假设这些手写体数据满足多元正态分布，即

$$p(X|C) = f_{\mathbf{x}}(x_1, \ldots, x_n) = \frac{1}{\sqrt{(2\pi)^n |\mathbf{\Sigma}|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}} \mathbf{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})\right)$$

其中, $\boldsymbol{\mu}$ 是多维变量的均值，$\mu_i = E(X_i)$, $\mathbf{\Sigma}$ 是协方差矩阵，$|\mathbf{\Sigma}|$ 是协方差矩阵的行列式。

$$\Sigma = \mathrm{E}\left[(\mathbf{X} - \mathrm{E}[\mathbf{X}])\,(\mathbf{X} - \mathrm{E}[\mathbf{X}])^T\right]$$

$$= \begin{bmatrix} \mathrm{E}[(X_1-\mu_1)(X_1-\mu_1)] & \mathrm{E}[(X_1-\mu_1)(X_2-\mu_2)] & \cdots & \mathrm{E}[(X_1-\mu_1)(X_n-\mu_n)] \\ \mathrm{E}[(X_2-\mu_2)(X_1-\mu_1)] & \mathrm{E}[(X_2-\mu_2)(X_2-\mu_2)] & \cdots & \mathrm{E}[(X_2-\mu_2)(X_n-\mu_n)] \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{E}[(X_n-\mu_n)(X_1-\mu_1)] & \mathrm{E}[(X_n-\mu_n)(X_2-\mu_2)] & \cdots & \mathrm{E}[(X_n-\mu_n)(X_n-\mu_n)] \end{bmatrix}$$

由于在计算时，很多概率相乘可能会导致很小的值出现，因此我们并不是直接利用上面的概率来计算，而是用取对数之后的结果来表示，即

$$p(y|x) \propto p(x_i|y) = \frac{1}{\sigma_y \sqrt{2\pi}} e^{-\frac{(x_i-\mu_y)^2}{2\sigma_y^2}} \quad i = 1, 2, 3, 4$$

最后就是要计算

$$\begin{aligned}\ln p(c_k|X) &= \ln p(X|c_k) + \ln p(c_k), \quad k = 1, 2, \cdots, L. \\ &= \ln \left(\frac{1}{\sqrt{(2\pi)^n |\mathbf{\Sigma}_k|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^{\mathrm{T}} {\mathbf{\Sigma}_k}^{-1} (\mathbf{x} - \boldsymbol{\mu}_k) \right) \right) + \ln p(c_k) \\ &= -\frac{n}{2} \ln 2\pi - \frac{1}{2} \ln |\mathbf{\Sigma}_k| - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^{\mathrm{T}} {\mathbf{\Sigma}_k}^{-1} (\mathbf{x} - \boldsymbol{\mu}_k) + \ln p(c_k)\end{aligned}$$

在实际计算中，测试数据中的各个类别的分别比较均匀，上式中有的项对于最后的结果影响较小，可以不用计算，最后就只需计算

$$\ln p(c_k|X) = -\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^{\mathrm{T}} {\mathbf{\Sigma}_k}^{-1} (\mathbf{x} - \boldsymbol{\mu}_k) + \ln p(c_k)$$

在代码的实现中，我们主要调用numpy中提供的各种线性代数的函数来计算矩阵相关的运算，特别要注意的是在计算协方差矩阵时，我们在原来的基础上，在对角线上加了一个很大的数λ, 这里我们测试后发现取15000效果最好（如下表所示,注意表中的运行时间与最后实际单独运行时的时间有区别，可能是由于这里是连续测试，很多数在内存中已经存在的缘故），这样做的原因是为了防止求协方差矩阵的逆时可能出现奇异的情况。

λ	Error rate	Run time(unit:s)
1000	6.21%	38.86
5000	4.57%	37.96
10000	4.17%	38.47
12000	4.14%	38.12
15000	4.06%	38.09
18000	4.16%	38.15

还有就是计算矩阵的行列式耗费时间特别长，含有协方差矩阵行列式那一项对最终结果的影响比较小，因此在求解时我们就直接忽略了, 这样做提高了计算的速度，具体可以看下面的示例代码，实际代码在文件mnist_nb_multi_Gaussian.py 中。

```
#!/usr/bin/env python
import numpy as np

# Load train and test data
train_image = np.fromfile('train-images.idx3-ubyte', dtype=np.uint8)
train_image = (train_image[16:]).reshape([60000,784])

train_label = np.fromfile('train-labels.idx1-ubyte', dtype=np.uint8)
train_label = (train_label[8:])

test_image = np.fromfile('t10k-images.idx3-ubyte', dtype=np.uint8)
test_image = (test_image[16:]).reshape([10000,784])

test_label = np.fromfile('t10k-labels.idx1-ubyte', dtype=np.uint8)
test_label = (test_label[8:])

# Calculate the A prioir probablity
p_y = np.array([train_label[train_label == i].shape[0]*1.0 / train_label.shape[0] for i in
    range(10)])
#print p_y

# Calculate the sigma and mu
sigma = {}
mu = {}
invsigma ={}
lambda = 15000
for i in range(10):
    dat = train_image[train_label == i, :]
    mu[i]    = np.average(dat, axis=0)
    # sigma is singular, so we add some large value on the diagnoal
    sigma[i] = np.cov(dat, rowvar=0) + lambda*np.eye(784, dtype=np.uint8)
    invsigma[i] = np.linalg.inv(sigma[i])

idx = np.zeros(10000, dtype=np.uint8)

for k in range(10000):
    kdat = test_image[k,:]
```

```
    gx = np.zeros(10)
    for j in range(10):
        gx[j] = -0.5*(kdat - mu[j]).dot(invsigma[j]).dot((kdat - mu[j]).T) + np.log(p_y[j])
    idx[k] = np.argmax(gx)

err = 1 - sum(idx == test_label)*1.0/10000
print "Error rate is " + str(err*100)+ "%."
```

下面是程序运行的结果，误差率为4.06%, 通过ipython测试耗时为1min 14s. 可以看出，这种方法比以上所有的Naive Bayes方法的准确率都高，但是计算时间也比较长。

```
Error rate is 4.06%.
CPU times: user 1min 10s, sys: 4.31 s, total: 1min 14s
Wall time: 43 s
```

3 基于SVM的手写体识别

3.1 SVM算法简介

SVM（support vector machine）从线性可分情况下的最优分类面发展而来。最优分类面就是要求分类线不但能将两类正确分开,且使分类间隔最大。SVM 考虑寻找一个满足分类要求的超平面,并且使训练集中的点距离分类面尽可能的远,也就是寻找一个分类面使它两侧的空白区域(margin)最大。过两类样本中离分类面最近的点且平行于最优分类面的超平面上的训练样本就叫做支持向量。margin越大，分类器有更强的纠错能力，也更稳定、更通用。

数学的层面，SVM即为求解一个二次规划的优化问题。在求取有约束条件的优化问题时，拉格朗日乘子法（Lagrange Multiplier) 和KKT条件是非常重要的两个求取方法，对于等式约束的优化问题，可以应用拉格朗日乘子法去求取最优值；如果含有不等式约束，可以应用KKT条件去求取。这两个方法求得的结果只是必要条件，只有当是凸函数的情况下，才能保证是充分必要条件。KKT条件是拉格朗日乘子法的泛化。总的来说，其算法包括：

(1) 选择核函数(将特征进行从低维到高维的转换，事先在低维上进行计算，而将实质上的分类效果表现在了高维上，避免了直接在高维空间中的复杂计算)；

(2) 选择参数（引入松弛变量处理有噪音数据，控制目标函数中两项：“寻找margin 最大的超平面”和“保证数据点偏差量最小”之间的权重）

(3) 求解二次规划的优化问题；

(4) 由支持向量构造判别函数。

3.2 代码实现

Step 1 载入sklearn 库函数

```
# mnist_svm by using scikit-learn
import numpy as np
from sklearn import svm
```

Step 2 加载分类数据（训练数据和测试数据）

```
# train examples: 60000, size: 28*28
train_image = np.fromfile('train-images.idx3-ubyte', dtype=np.uint8)
train_image = (train_image[16:]).reshape([60000,28*28])

train_label = np.fromfile('train-labels.idx1-ubyte', dtype=np.uint8)
train_label = (train_label[8:])
```

```
# test examples: 10000, size: 28*28
test_image = np.fromfile('t10k-images.idx3-ubyte', dtype=np.uint8)
test_image = (test_image[16:]).reshape([10000,28*28])

test_label = np.fromfile('t10k-labels.idx1-ubyte', dtype=np.uint8)
test_label = (test_label[8:])
```

Step 3 选择小规模数据检验各类分类器性能。由于分类用时长，提取训练和测试数据的十分之一。由于测试数据中，前5000个数据比较清晰，后5000个数据比较模糊，因此选择范围是中间数据段。

```
# select smaller train sample: 6000, size: 28*28
strain_image = train_image[0:6000,:]
strain_label = train_label[0:6000]
# select smaller test sample: 1000= former500 + later500, size: 28*28
stest_image = test_image[4500:5500,:]
stest_label = test_label[4500:5500]
```

Step 4 选择分类函数，比较不同条件下的分类结果

```
# -- choose differnt functions to fit model -- # -- change kernel-- change parameters
    ----#
#model = svm.LinearSVC()                          # choose lineaar condition
#model = svm.NuSVC(kernel="poly",degree=2)        # choose Non-lineaar condition
#model = svm.SVC(kernel="linear")                 # create a svm with linear kernel
#model = svm.SVC(kernel="rbf",gamma=2)            # create a svm with rbf kernel
model = svm.SVC(kernel="poly",coef0=0.0,degree=2,gamma=0.0) # create a svm with
    polynomial kernel
#model = svm.SVC(kernel="sigmoid",coef0=0.0,gamma=1)     # create a svm with sogmoid
    kernel
```

Step 5 进行分类计算

```
# train, fit the model
model.fit(train_image, train_label)

# test
svm_class = model.predict(test_image)
```

Step 6 统计错误率

```
# error amount
error_count = 0
for i in xrange(test_label.shape[0]):
    if svm_class[i] !=  test_label[i]:
        error_count += 1

# error rate
errorRate = float(error_count) / float(test_label.shape[0])
```

3.3 试验结果

选择不同的核函数以及其不同参数时，识别的精度会有变化(如下表所示), 具体来说，SVM中核函数的类型主要有:

- linear: $< x, x' >$
- polynomial: $(\gamma < x, x' > +r)^d$, d is specified by keyword $degree$, r is by $coef0$.
- rbf: $exp(-\gamma|x, x'|^2)$, γ is specified by keyword $gamma$, must be greater than 0.
- sigmoid: $\tanh(\gamma < x, x' > +r)$, where r is specified by $coef0$.

kernel	$coef0$	$degree$	$gamma$	Error Rate
LinearSVC	-	-	-	0.131
	-	-	-	0.148
	-	-	-	0.135
NuSVC	-	-	-	0.881
NuSVC(kernel="poly")	-	-	-	0.12
SVC(kernel="linear")	-	-	-	0.078
	-	-	-	0.078
	-	-	-	0.078
SVC(kernel="poly")	0.1	3	-	0.04
	0.2	3	-	0.04
	1	3	-	0.04
	-	1	-	0.078
	-	2	-	0.034
	-	-	-	0.04
	-	4	-	0.05
	-	5	-	0.064
	-	-	0.5	0.04
	-	-	1	0.04
	-	-	2	0.04
SVC(kernel="rbf")	-	-	-	0.881
	-	-	0.5	0.881
	-	-	2	0.881
SVC(kernel="sigmoid")	-	-	-	0.881
	0.5	-	-	0.881
	-	-	1	0.881

(上表为不同Kernel和参数对错误率的影响；小规模数据结果，- 默认值，存在相应参数的情况下，$coef0 = 0.0$, $degree = 3, gamma = 0.0$)

通过测试比较，发现在model = svm.SVC(kernel="poly",coef0=0.0,degree=2,gamma=0.0)情况下，分类错误率最低。用它测试大规模数据，错误率为0.0194。

```
>>> errorRate
0.0194
```

3.4 SVM小结

本例手写数字体识别，线性和多项式的SVM的分类效果更好。以二次多项式为最佳。高斯Gauss径向基函数则是局部性强的核函数，但是全局性较差；而Sigmoid函数作为核函数时，支持向量机实现的就是一种多层感知器神经网络，应用SVM方法，隐含层节点数目(它确定神经网络的结构)、隐含层节点对输入节点的权值都是在设计(训练)的过程中自动确定的。而且支持向量机的理论基础决定了它最终求得的是全局最优值而不是局部最小值，也保证了它对于未知样本的良好泛化能力而不会出现过学习现象，但是在本例中分类效果较差。

4 总结

通过kNN,Navie Bayes和SVM分类方法对MNIST手写数字进行识别，发现用knn算法（当k=3,采用欧式距离）时，错误率最小可达到2.83%，但运行时间较长为11min02s；基于多元正态分布的Navie Bayes算法，当λ取15000时，分类错误率为4.06%，且运行时间仅1min14s；SVM 算法在$model = svm.SVC(kernel = "poly", coef0 = 0.0, degree = 2, gamma = 0.0)$ 情况下，分类错误率最低为1.94%。显然以准确率为标准时，最优模型为最后的SVM 方法，参数选择如上所列。