《人工智能与深度学习》课程

实验报告

学号：04022212

姓名：钟 源

2022 年 04 月 09 日

实验三：支持向量机

一、实验目的

了解支持向量机（Supported Vector Machine） 的基本概念及原理，了解核函数的基本原理以及作用。通过实验学会 SVM 在简单分类问题中的应用并通过 SVM 构建一个垃圾邮件分类系统。

二、实验内容

完成 SVM 对给定数据集的分类问题。 注意给出实验代码和详细过程介绍。

1. 实验代码补全

1）高斯核函数：

```
# 函数：高斯核函数
def gaussian_kernel(x1, x2, sigma):
    # your code here
    return np.exp(-sum((x1 - x2) ** 2.0) / (2 * (sigma ** 2.0)))
```

2）线性可分 SVM（步骤 3）：

```
    # --------------- 步骤 3 -----------------
    # 训练线性 SVM（C = 100）
```

```
    linear_svm: SVC = svm.SVC(C=100, kernel='linear')
    linear_svm.fit(X, y.ravel())

    # 绘制 C=100 的 SVM 决策边界
    # your code here  1）
    plt.title('C=100 的 SVM 决策边界')
    plot(np.c_[X, y])
    visualize_boundary(X, linear_svm)
    plt.show(block=True)
```

3）非线性可分 SVM（步骤 2）：

```
# --------------- 步骤 2 -----------------
    # 加载数据集 2
    mat = scipy.io.loadmat("dataset_2.mat")
    X, y = mat['X'], mat['y']

    # 绘制数据集 2
    plt.title('数据集 2 分布')
    plot(np.c_[X, y])
    plt.show(block=True)
    # 训练高斯核函数 SVM
    sigma = 0.01
    rbf_svm = svm.SVC(C=1, kernel='rbf', gamma=1.0 / sigma)  # gamma is actually
inverse of sigma
    rbf_svm.fit(X, y.ravel())

    # 绘制非线性 SVM 的决策边界
    plt.title('非线性 SVM 的决策边界')
    plot(np.c_[X, y])
    visualize_boundary(X, rbf_svm)
    plt.show(block=True)
```

4）最优参数搜索（设定模型的 C 和 gamma 值）：

```
# 通过 rbf_svm.set_params 可设定模型的 C 和 gamma 值
    y = y.ravel()

    best = params_search(X, y, X_val, y_val)
    rbf_svm.set_params(C=best['C'])
    rbf_svm.set_params(gamma=best['gamma'])
    rbf_svm.fit(X, y)
```

5）垃圾邮件分类（对垃圾邮件进行预测）：

```
def part_4():
    mat = scipy.io.loadmat("spamTrain.mat")
```

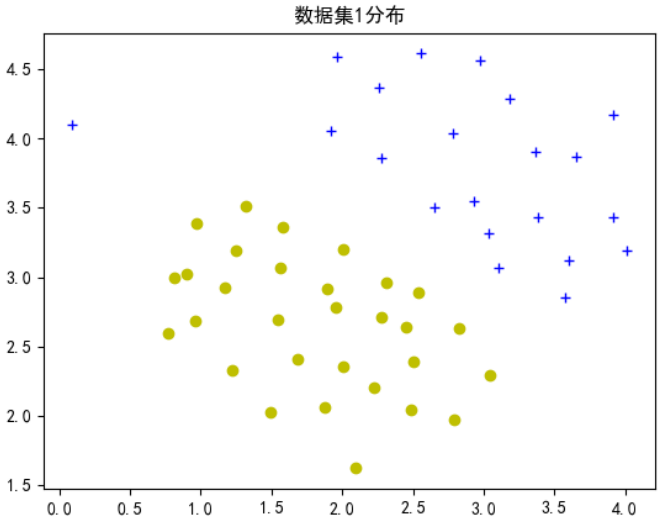
```
X, y = mat['X'], mat['y']

# linear_svm = pickle.load(open("linear_svm.svm", "rb"))
linear_svm = svm.SVC(C=0.1, kernel='linear')
linear_svm.fit(X, y.ravel())
# pickle.dump(linear_svm, open("linear_svm.svm", "wb"))

word_indices, _ = email_preprocess('spamSample2.txt')
features = feature_extraction(word_indices).transpose()

print(linear_svm.predict(features))
```

2. 实验过程与结果

2.1 线性可分 SVM：

1）加载并绘制数据集 1：

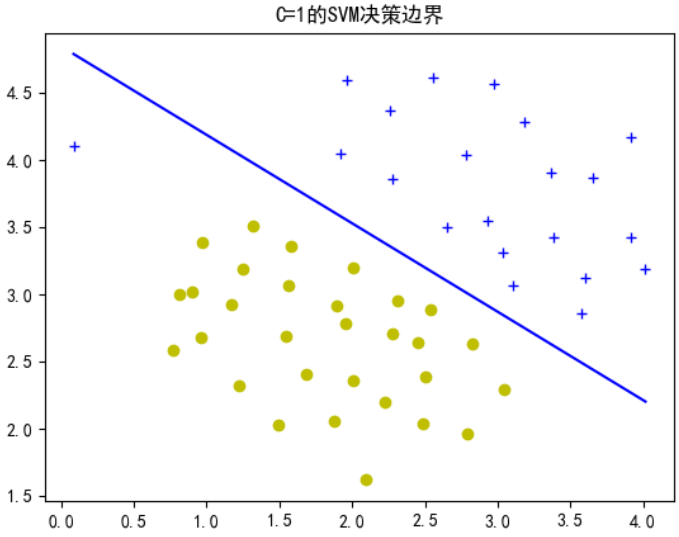
在训练模型之前首先需要对数据集进行加载以及观察，绘制结果如下图所示：



可以看到：数据集 1 是一个简单的二维数据集，并且可以清晰地看到两类数据点之间存在着明显的间隔，在图像的最左边有一个偏离总体很远的正样本离群点，离群点对模型的拟合往往会有一定的影响作用。下一步尝试给 SVM 模型的 C 参数赋予不同值以观察不同的拟合效果。

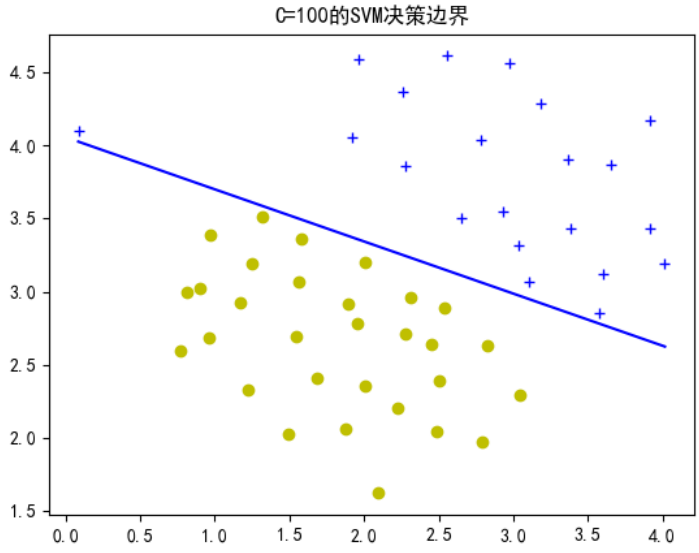
2）设置 C=1 时训练模型：

首先设置 C=1，训练 SVM 模型，绘制该 SVM 拟合出的决策边界如下图所示：



3）设置 C=100 时训练模型：

接着设置 C=1，训练 SVM 模型，绘制该 SVM 拟合出的决策边界如下图所示：



2.2 非线性可分 SVM：

1）计算相似度距离

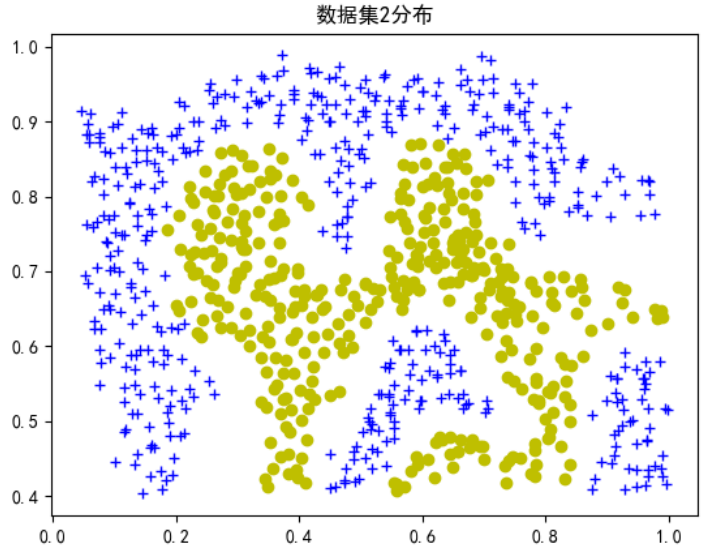
首先基于公式$K(x^i,x^j)=e^{\left(\frac{|x^i-x^j|^2}{2\sigma^2}\right)}=e^{-\frac{\sum_{k=1}^{n}\left(x_k^i-x_k^j\right)^2}{2\sigma^2}}$实现高斯核函数返回两样本之间的相似度距离，这部分已经在之前代码补全部分给出。

然后根据高斯核函数计算给定的两个样例 x1 和 x2 的相似度来测试函数，结果如下：

```
Python 控制台
Backend MacOSX is interactive backend. Turning interactive mode on.
样本x1和x2之间的相似度: 0.324652
```

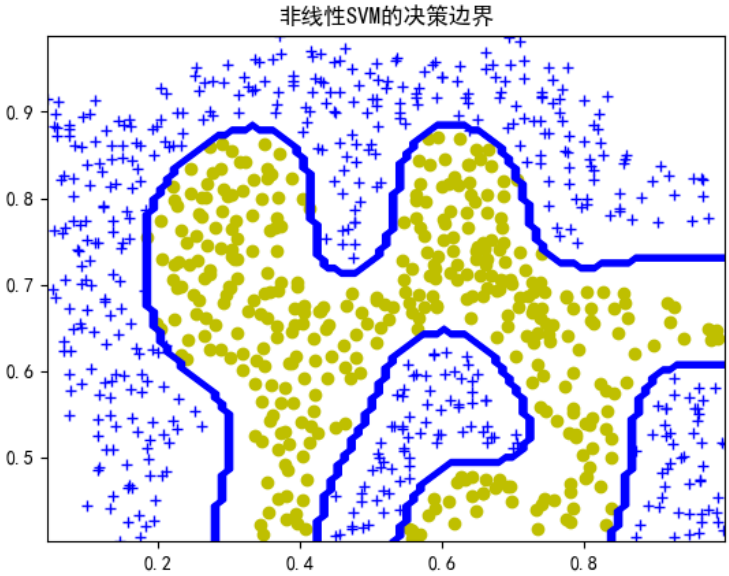
2）加载并绘制数据集 2：

接下来将需要加载并绘制数据集 2，用于非线性 SVM 的训练，结果如下图所示：



3）训练模型并绘制决策边界：

从图中可以观察到正样本与负样本之间并不存在线性划分边界，但是通过基于高斯核函数的 SVM 可以拟合出非线性的决策边界，结果如下图所示：

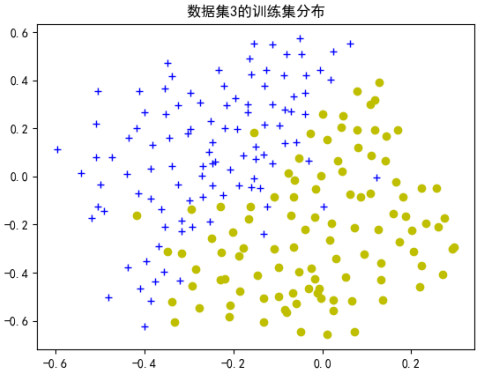


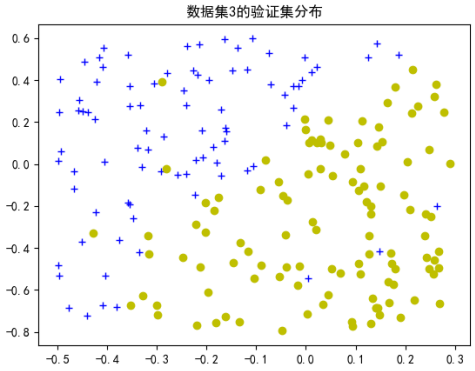
可以观察到在图中，该决策边界可以将绝大部分正样本和负样本划分开。

2.3 最优参数搜索：

1）加载数据集 3 的训练集和验证集：

在数据集 3 中，数据被划分为了训练集以及验证集，同样通过绘制训练集和验证集以观察数据，结果如下图所示：



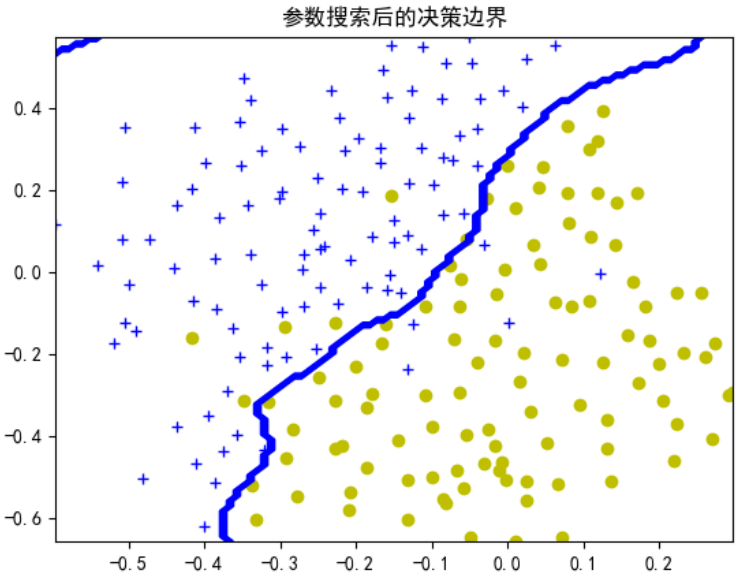


2）参数搜索：

基于训练集训练的模型将在验证集上进行测试并根据验证集的反馈结果对模型参数进行微调。并且通过验证集对 SVM 模型的 C 参数和 σ 参数进行调整，设置的搜索范围为[0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]，在 params_search 函数中将在这两个参数上分别使用这 8 个值进行测试，具有最小误差的一组参数则为最优的参数。

3）绘制决策边界：

在确定了最优的 C 值与 σ 值之后，使用最优参数组合对模型进行训练并观察其拟合的决策边界，结果如下图所示：



2.4 垃圾邮件分类：

1）邮件预处理：

首先对各个邮件样本进行文本预处理、特征提取、设定标签（是否为垃圾邮件）。在对文本进行预处理之前将实现一个对给定的词典文件 vocab.txt 中的词汇进行标号的函数 vocaburary_mapping，在该函数中，每个词汇都被映射到了一个序号，这将便于之后的文本特征提取。

完成函数后，基于实验手册提到的预处理规则对输入的邮件文本进行处理，并将预处理过后的邮件文本中每个词汇对应于词典中词汇的序号进行标注。接下来将对 emailSample1.txt 和 emailSample2.txt 进行预处理。

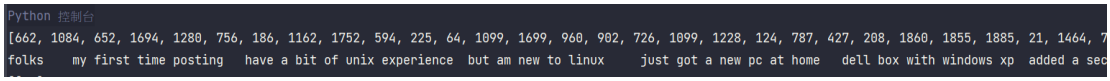
对 emailSample1.txt 预处理的结果如下：

具体如下：

[86, 916, 794, 1077, 883, 370, 1699, 790, 1822, 1831, 883, 431, 1171, 794, 1002, 1893, 1364, 592, 1676, 238, 162, 89, 688, 945, 1663, 1120, 1062, 1699, 375, 1162, 479, 1893, 1510, 799, 1182, 1237, 810, 1895, 1440, 1547, 181, 1699, 1758, 1896, 688, 1676, 992, 961, 1477, 71, 530, 1699, 531]

anyone knows how much it costs to host a web portal well it depends on how many visitors you re expecting this can be anywhere from less than number bucks a month to a couple of dollarnumber you should checkout httpaddr or perhaps amazon ecnumber if youre running something big to unsubscribe yourself from this mailing list send an email to emailaddr

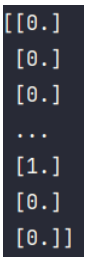
对 emailSample2.txt 预处理的结果如下（具体数据太长没有展现）：



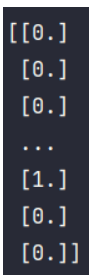
2）邮件特征提取：

在这一部分，需要实现对经过预处理后的邮件进行特征提取，即将邮件映射成一个 n 维向量 **x**，其中 n 为词典的词汇量，每一维对应词典中的一个词汇。在 **x** 中$x_i \in \{0,1\}, 1 \le i \le n$，即对于第 i 个词汇，如果其出现在了邮件中，则有$x_i = 1$，否则$x_i = 0$。

对 emailSample1.txt 的特征提取如下图所示：



对 emailSample2.txt 的特征提取如下图所示：



3）SVM 分类：

接下来将基于一个已经预处理过数据的训练集 spamTrain.mat 进行 SVM 模型训练，该训练集包含 4000 条垃圾邮件和正常邮件的训练样本，测试集 spamTest.mat 包含 1000 条测试样本。

训练结果如下图所示， 训练集的准确率达到了 99.8%， 测试集的准确率达到了 98.9%

```
Accuracy rate for training set:
99.825%
Accuracy rate for testing set:
98.9%
```

查看训练得到的 SVM 对各个词汇的权重系数来找出权重最高的词汇，结果如下所示，当一封邮件包含如下单词时则会使得邮件有更高的概率被识别为垃圾邮件。

```
spam
that
urgent
wrong
datapow
linux
round
numberth
useless
unsubscrib
august
ratio
xp
toll
http
```

接下来对给定的两封垃圾邮件样例进行预测， 发现对于 spamSample2.txt 的预测结果为 y=1，但对于 spamSample1.txt 的预测结果为 y=0，这说明该 SVM 也许还有一些改进的空间。

```
[0]
[1]
```

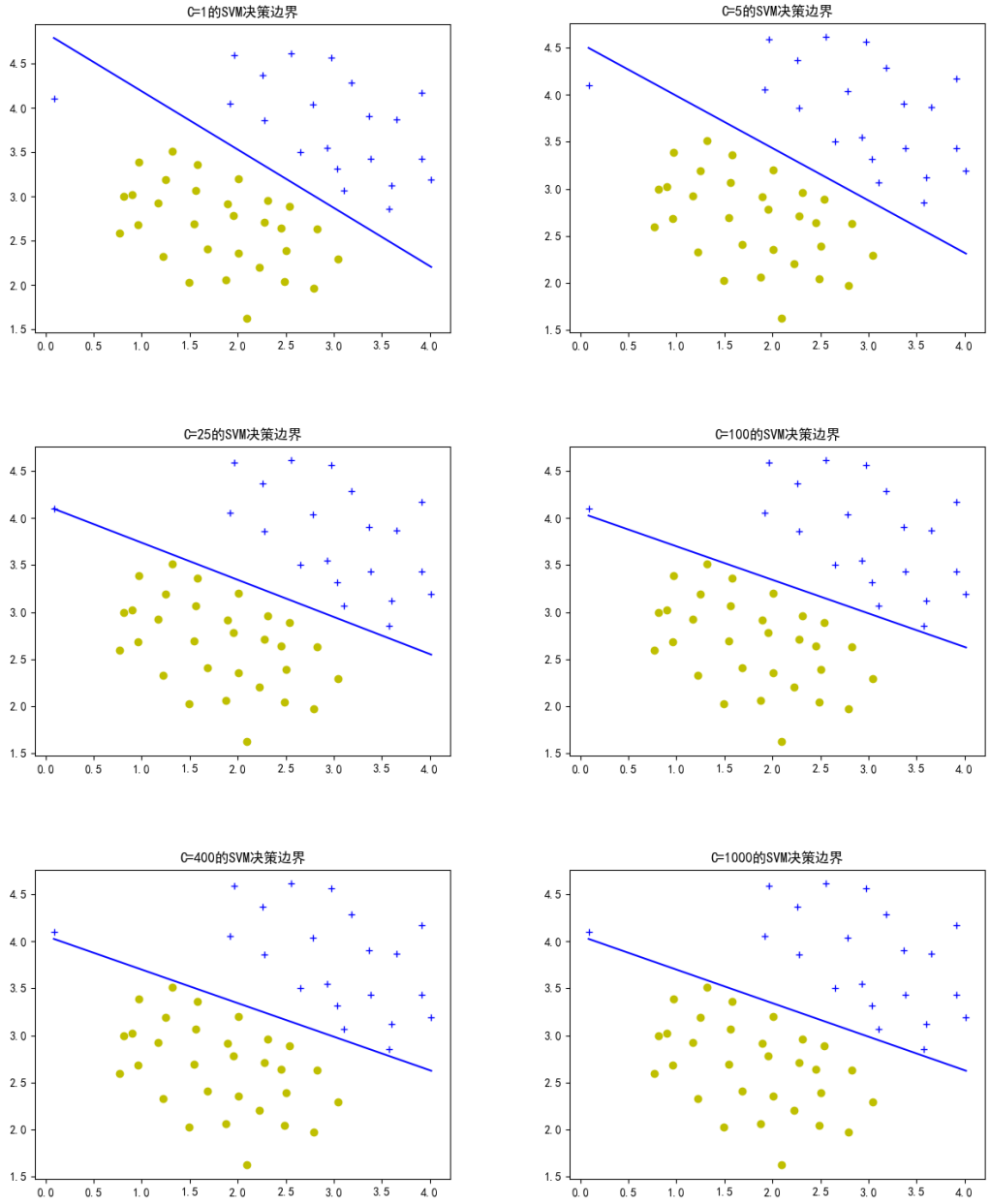
三、提高训练

基于实验手册的内容，尝试对现有实验做修改和调整（本项内容可选）。比较修改后对 SVM 的性能影响，分析猜测其背后的原因。

1. 调整“线性可分 SVM”中的 C 值：

C 参数是调节对模型错误分类的惩罚的一个参数，越大的 C 参数则意味着模型需要更加精确地拟合样本以保证惩罚达到最小。尝试调节 C 参数，选取几个

代表性图像如下：



可以看到当 C 从 1 变化到 100 时，决策边界持续向下方偏转，值得注意的是，当 C=25 时，离群点恰好在决策边界上在。当 C 增大到 100 之后，继续增大 C ，线性 SVM 的决策边界不再发生明显变化

不过过大的 C 值对于 SVM 决策边界的普适性并不好，这是因为在决策边界附近的样本点，尤其是支持向量，一旦加上微弱的噪声，就容易出现错判的情况。

2. 调整“非线性可分 SVM”中的 sigma 值：

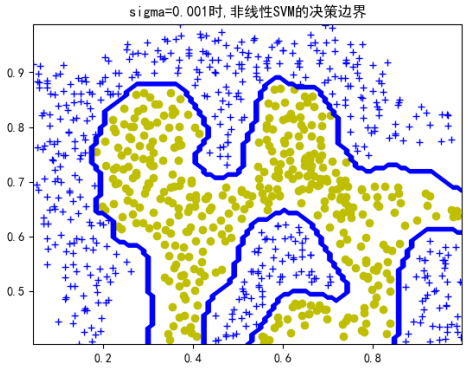
接着尝试探究在非线性可分 SVM 中，不同的高斯核函数的参数 sigma 对于决策边界的影响，同时为了更量化地看到影响，编写函数呈现训练集中分类发错误率，实现代码如下：

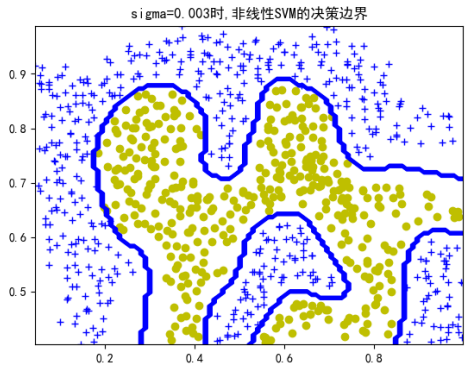
```
def train_sigma(X,y,sigma):
    # 训练高斯核函数 SVM
    rbf_svm = svm.SVC(C=1, kernel='rbf', gamma=1.0 / sigma)  # gamma is actually
inverse of sigma
    rbf_svm.fit(X, y.ravel())

    # 查看训练集的错误率：
    predictions = []
    m = np.shape(X)[0]
    for i in range(0, m):
        prediction_result = rbf_svm.predict(X[i].reshape(-1, 2))
        predictions.append(prediction_result[0])
    # sadly if you don't reshape it, numpy doesn't know if it's row or column vector
    predictions = np.array(predictions).reshape(m, 1)
    error = (predictions != y.reshape(m, 1)).mean()
    print('When sigma={}, error of training set:'.format(sigma))
    print(error)

    # 绘制非线性 SVM 的决策边界
    plt.title('sigma={}时,非线性 SVM 的决策边界'.format(sigma))
    plot(np.c_[X, y])
    visualize_boundary(X, rbf_svm)
    plt.show(block=True)
```

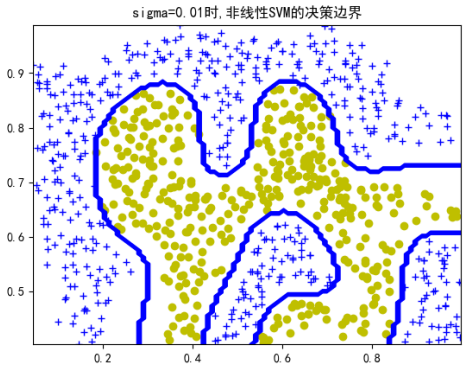
分别取 sigma 为 0.001、0.003、0.01、0.03、0.1、1，得到的训练集错误率如下所示：

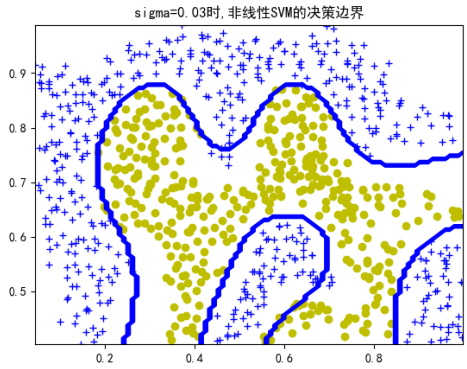
```
When sigma=0.001, error of training set:
0.0
When sigma=0.003, error of training set:
0.002317497103128621
When sigma=0.01, error of training set:
0.002317497103128621
When sigma=0.03, error of training set:
0.02085747392815759
When sigma=0.1, error of training set:
0.11587485515643106
When sigma=1, error of training set:
0.19119351100811124
```

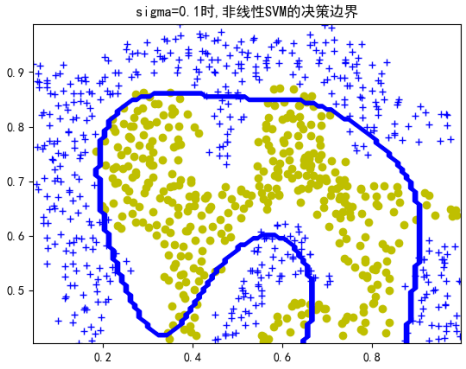
绘制的决策边界如下：

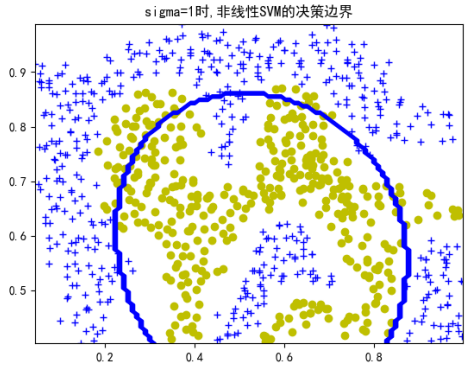












可以看，当 sigma 较大时（如 sigma=1），SVM 分类效果比较糟糕，决策边界轮廓较模糊。随着 sigma 的不断减小，SVM 分类效果越来越好，错误率逐渐降低。结合高斯分布的方差参数分析，当高斯分布方差参数 sigma 越大时，峰值越不突出，在坐标轴上的分布范围越大，属于某一类的可能性越混淆，各点概率也比越小。而 sigma 越小，有越大的概率准确确定某个值属于哪个高斯分布。推测高斯核函数 sigma 参数也是如此，这点从错误率随 σ 的变化可以看出。

然而，sigma 也并非越小越好——虽然数据集 2 没有测试集，但是仍然可以判断当 sigma=0.001 时 SVM 模型出现了过拟合。这是因为当从 sigma 从 0.01 减小到 0.003 后训练集的错误率没有下降，而继续下降到 0.001 后错误率就变成 0 了。从图中也可以看出，模型学习能力过强，决策边界与数据拟合得太精细，缺乏泛化能力。

3. 进一步搜索最优参数：

在“最优参数搜索”中，已经通过在建议值之间比较而得到了在数据集 3 的参考范围内最优参数：C=0.3、sigma=0.01，错误率为 0.035。

接下来借鉴 BCD 算法（块坐标下降法）的思想，交替优化 C 和 sigma 值，具体优化方法参考了二分法。

核心函数代码如下：

```
def train(X, y, X_val, y_val, C, sigma):
    raveled_y = y.ravel()
    m_val = np.shape(X_val)[0]
    m = np.shape(X)[0]
    # train the SVM first
    rbf_svm = svm.SVC(kernel='rbf')
    rbf_svm.set_params(C=C)
    rbf_svm.set_params(gamma=1.0 / sigma)
    rbf_svm.fit(X, raveled_y)
    # test it out on validation data
    predictions = []
    for i in range(0, m_val):
        prediction_result = rbf_svm.predict(X_val[i].reshape(-1, 2))
        predictions.append(prediction_result[0])
    predictions = np.array(predictions).reshape(m_val, 1)
    val_error = (predictions != y_val.reshape(m_val, 1)).mean()
    # test it out on validation data
    predictions = []
    for i in range(0, m):
        prediction_result = rbf_svm.predict(X[i].reshape(-1, 2))
        predictions.append(prediction_result[0])
    predictions = np.array(predictions).reshape(m, 1)
    train_error = (predictions != y.reshape(m, 1)).mean()
    return val_error, train_error


def params_search_C(X, y, X_val, y_val, last_best1, last_C_sec):
    c1 = last_best1['C']
    c2 = last_C_sec
    if c1 > c2:
        temp = c1
        c1 = c2
        c2 = temp
    sigma = last_best1['sigma']
    np.c_values = [c1, (c1 + c2) / 2.0, c2]
    best1 = last_best1
    val_error = [0, 0, 0]
```

```
    train_error = [0, 0, 0]
    for i in range(3):
        val_error[i], train_error[i] = train(X, y, X_val, y_val, np.c_values[i],
sigma)
    [error1, error2], [i1, i2] = find_smallest_two(val_error)
    best1['val_error'] = error1
    best1['train_error'] = train_error[i1]
    best1['C'] = np.c_values [i1]
    C_sec = np.c_values [i2]
    return best1, C_sec

def params_search_sigma(X, y, X_val, y_val, last_best1, last_sigma_sec):
    s1 = last_best1['sigma']
    s2 = last_sigma_sec
    if s1 > s2:
        temp = s1
        s1 = s2
        s2 = temp
    C = last_best1['C']
    sigma_values = [s1, (s1 + s2) / 2.0, s2]
    best1 = last_best1
    train_error = [0, 0, 0]
    val_error = [0, 0, 0]
    for i in range(3):
        val_error[i], train_error[i] = train(X, y, X_val, y_val, C, sigma_values[i])
    [error1, error2], [i1, i2] = find_smallest_two(val_error)
    best1['val_error'] = error1
    best1['train_error'] = train_error[i1]
    best1['sigma'] = sigma_values[i1]
    best1['gamma'] = 1.0 / best1['sigma']
    sigma_sec = sigma_values[i2]
    return best1, sigma_sec
```

设定优化次数为 9 轮，运行结果如下

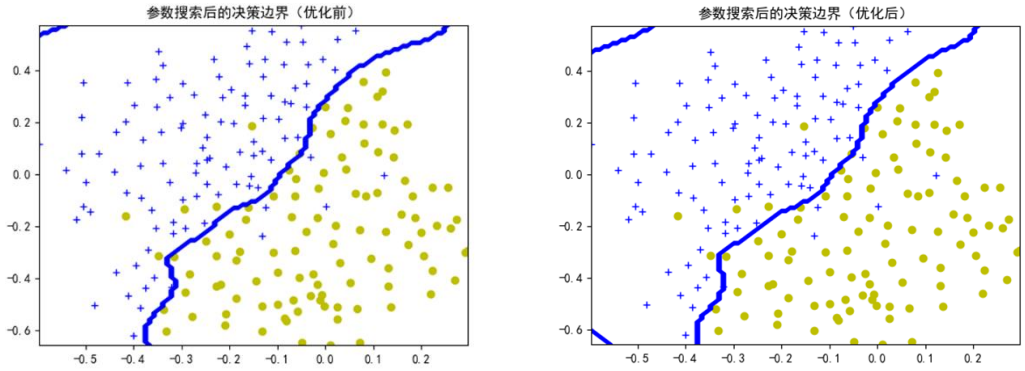
```
Before Improve,
{'C': 0.3, 'sigma': 0.01, 'val_error': 0.035, 'train_error': 0.04739336492890995, 'gamma': 100.0}

 Times 1 of C optimization,
val_error:0.03; train_error:0.05687203791469194; C:0.2; sigma:0.01;
C_sec:0.3

 Times 1 of sigma optimization,
val_error:0.03; train_error:0.05687203791469194; C:0.2; sigma:0.01;
sigma_sec:0.02
```

……

```
 Times 9 of C optimization,
val_error:0.03; train_error:0.05687203791469194; C:0.2; sigma:0.01;
C_sec:0.20039062500000002

 Times 9 of sigma optimization,
val_error:0.03; train_error:0.05687203791469194; C:0.2; sigma:0.01;
sigma_sec:0.010039062500000001

 After Improve,
{'C': 0.2, 'sigma': 0.01, 'val_error': 0.03, 'train_error': 0.05687203791469194, 'gamma': 100.0}
```

可以看到，在优化了一轮之后，结果就收敛到了 C=0.2，sigma=0.01 了，之后虽然区间在不断缩小，但是结果上并没有实际变化，所以可以确定——C=0.2，sigma=0.01 至少是一个局部最优解，此时验证集的错误率为 0.03，训练集需错误率为 0.057。

优化前后的绘制的决策边界对比如下：



可以看到，对比改进前，决策边界包括的范围确实会更小，使得分类更准确了一些，但是改变并不太大，说明前期设置参考范围比较合理，优化的必要性不是特别大。

4. 使用多项式核函数：

下一步尝试更换不同的核函数，查看对于模型分类的影响。

查阅资料可知， SVM 常用的核函数为线性核函数、 sigmoid 核函数、高斯核函数、多项式核函数等。线性函数在之前已经有过相关实验，故而不再重复，而使用 sigmoid 核函数 SVM 某种意义上就变成了多层感知器（神经网络）。因此这里选取多项式核函数与高斯核函数对比，这样既能观察到线性核函数的结果，又能体会到核函数作为处理非线性 SVM 中发挥的重要作用以及 SVM 的工作过程。

首先运行之前在进一步搜索中确定的最优参数的高斯核函数，得到的结果如下图所示：

```
 After Improve,
{'C': 0.2, 'sigma': 0.01, 'val_error': 0.03, 'train_error': 0.05687203791469194, 'gamma': 100.0}
```

可以看出，当核函数为高斯核函数时，搜索到的最优参数为 C=0.2，σ=0.01，此时训练集准确率约为 94.31%，验证集准确率为 97.0%，没有发生过拟合。

1） 使用维度为 1 的多项式核函数：

对于多项式函数，搜索参数仍然为惩罚参 C 与倍乘常数 gamma（相当于高斯核函数里方差参数 sigma 的倒数），由于对多项式函数性质不太清楚，搜索范围设置为[0.001, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, 100]，并且仍然进行 BCD 算法优化参数。

首先设置多项式核函数维度为 1，效果接近线性核函数，并对 SVM 进行最优参数搜索，核心代码如下（其他更改类似）：

```
def poly_params_search(X, y, X_val, y_val,degree):
    np.c_values = [0.001, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, 100]
    sigma_values = [0.001, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, 100]
    best = {'C': 0.0, 'sigma': 0.0, 'val_error': 999, 'train_error': 999}

    raveled_y = y.ravel()
    m_val = np.shape(X_val)[0]
    m = np.shape(X)[0]

    poly_svm = svm.SVC(kernel='poly', degree=degree)

    for C in np.c_values:
        for sigma in sigma_values:
            # train the SVM first
            poly_svm.set_params(C=C)
            poly_svm.set_params(gamma=1.0 / sigma)
            poly_svm.fit(X, raveled_y)
            # 计算验证集错误率
            predictions = []
            for i in range(0, m_val):
                prediction_result = poly_svm.predict(X_val[i].reshape(-1, 2))
                predictions.append(prediction_result[0])
            predictions = np.array(predictions).reshape(m_val, 1)
            val_error = (predictions != y_val.reshape(m_val, 1)).mean()
            # 计算训练集错误率
            predictions = []
            for i in range(0, m):
                prediction_result = poly_svm.predict(X[i].reshape(-1, 2))
                predictions.append(prediction_result[0])
            predictions = np.array(predictions).reshape(m, 1)
            train_error = (predictions != y.reshape(m, 1)).mean()
            # get the lowest error
            if val_error < best['val_error']:
                best['val_error'] = val_error
                best['train_error'] = train_error
                best['C'] = C
```

```
                best['sigma'] = sigma
    best['gamma'] = 1.0 / best['sigma']

    return best
```

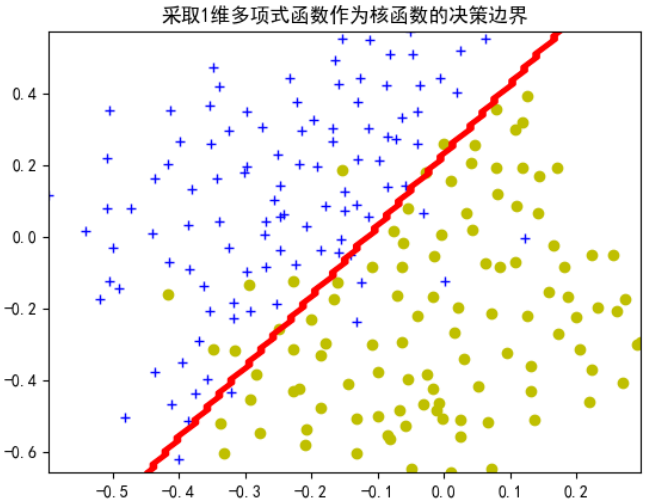
程序运行结果下所示：

```
Before Improve,
{'C': 0.1, 'sigma': 0.01, 'val_error': 0.065, 'train_error': 0.07582938388625593, 'gamma': 100.0}
```

……

```
 After Improve,
{'C': 0.0775, 'sigma': 0.01, 'val_error': 0.065, 'train_error': 0.08056872037914692, 'gamma': 100.0}
```

绘制的决策边界如下：



可以看到：决策边界近似于直线，与线性核函数有近似效果，分类情况比较良好。此时搜索到的最优参数为： C=0.0775，sigma=0.01，在最优参数的条件下训练集准确率为 90.05%，验证集准确率为 93.5%，没有发生过拟合。其准确率与高斯核函数相比，已经很高了，但是仍然差于高斯核函数。

2） 使用维度为 2 的多项式核函数：

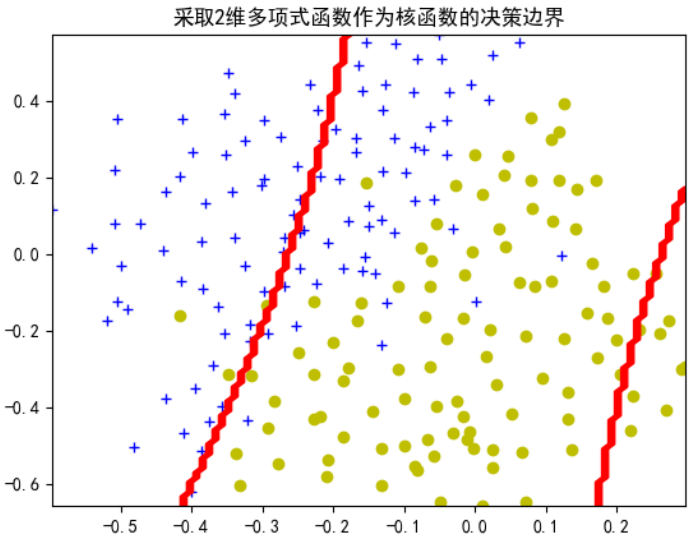
之后设置多项式核函数维度为 2，并对非线性可分的 SVM 进行最优参数搜索，运行结果如下：

```
Before Improve,
{'C': 0.01, 'sigma': 0.03, 'val_error': 0.285, 'train_error': 0.2985781990521327, 'gamma': 33.333333333333336}
```

……

```
 After Improve,
{'C': 0.006625, 'sigma': 0.03, 'val_error': 0.28, 'train_error': 0.3127962085308057, 'gamma': 33.333333333333336}
```

绘制的决策边界如下：



可以看到：决策边界的分类精度下降了很多，而且右半部分的曲线也缺乏存在的必要。此时搜索到的最优参数为： C=0.00625，sigma=0.03，在最优参数的条件下训练集准确率为 68.72%，验证集准确率为 72.0%。此时的训练效果远远差于一维多项式核函数与高斯核函数

3） 使用维度为 3 的多项式核函数：

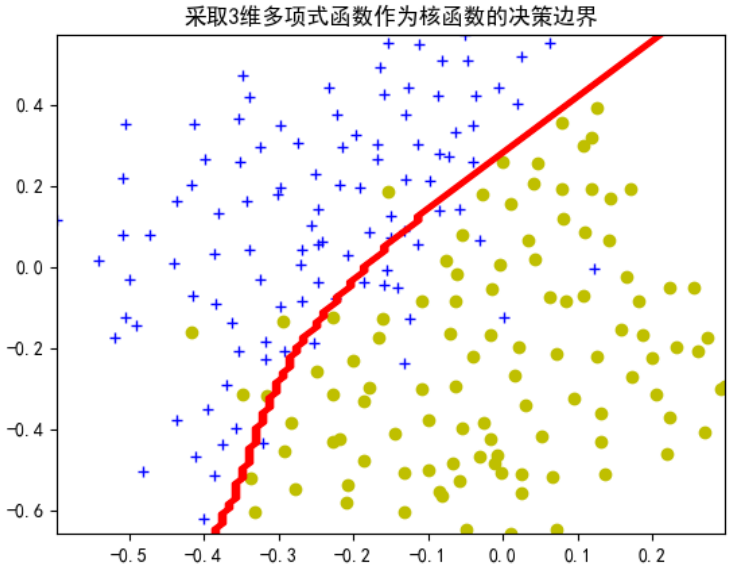
之后设置多项式核函数维度为 3，并对非线性可分的 SVM 进行最优参数搜索，运行结果如下：

```
Before Improve,
{'C': 0.03, 'sigma': 0.01, 'val_error': 0.07, 'train_error': 0.0947867298578199, 'gamma': 100.0}
```

……

```
 After Improve,
{'C': 0.055, 'sigma': 0.01, 'val_error': 0.07, 'train_error': 0.0947867298578199, 'gamma': 100.0}
```

绘制的决策边界如下：



从三维多项式核函数的决策边界可以看出，此时的分类效果不错。此时搜索到的最优参数为：C=0.055，sigma=0.01，在最优参数的条件下训练集准确率为91.52%，验证集准确率为 93.0%，没有发生过拟合。此时的训练效果远远优于二维多项式核函数，与一维多项式核函数相当，但仍不如高斯核函数。

4） 使用维度为 4 的多项式核函数：

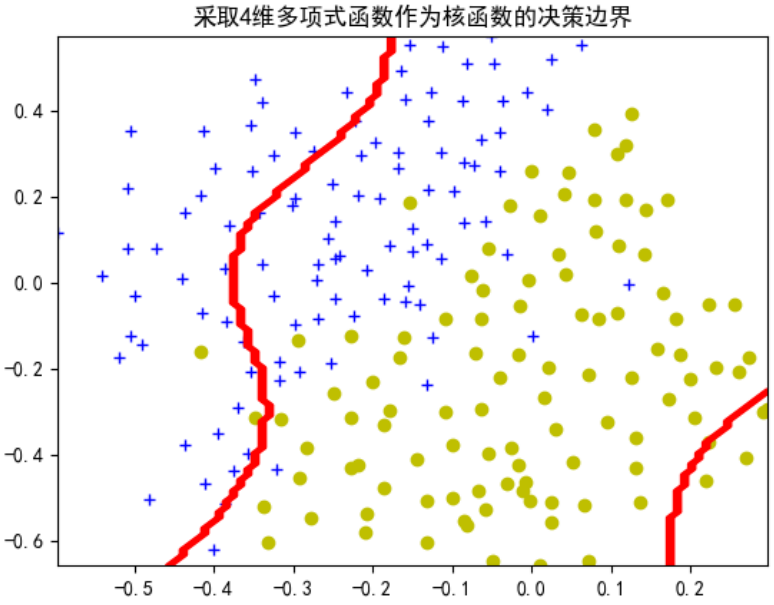
之后设置多项式核函数维度为 4，并对非线性可分的 SVM 进行最优参数搜索，运行结果如下：

```
Before Improve,
{'C': 30, 'sigma': 1, 'val_error': 0.3, 'train_error': 0.35545023696682465, 'gamma': 1.0}
```

……

```
After Improve,
{'C': 20.0, 'sigma': 1, 'val_error': 0.29, 'train_error': 0.33649289099526064, 'gamma': 1.0}
```

绘制的决策边界如下：



从四维多项式核函数的决策边界可以看出，此时的分类效果也非常不好。此时搜索到的最优参数为：C=20，sigma=1，在最优参数的条件下训练集准确率为66.35%，验证集准确率为 71.0%。而且可以注意到，到了四维之后，模型训练时间明显变长了很多。

5）不同核函数的对比：

当尝试把维度提升到五维，发现训练时间过长，在已经有较好的模型情况下，训练一个成本是其他模型数以百倍的模型并不划算，因此不再探究更高维度的多项式核函数。至于训练成本迅速增长的原因，推测为多项式核的维度随着次数的升高呈指数增长（虽然核技巧避免了显式计算高维特征，但核矩阵的计算仍可

能变得复杂）以及稀疏性降低（高次多项式可能生成大量非零权重，导致模型依赖过多的特征组合，难以保持稀疏性，从而影响效率与解释性）。

接着对比不停的核函数对于数据集 3 的分类效果，整理成表格如下：

核函数	最优 C 值	最优 sigma 值	训练集准确率	验证集准确率
高斯核函数	0.2	0.01	94.31%	97.0%
一维多项式	0.0775	0.01	90.05%	93.5%
二维多项式	0.00625	0.03	68.72%	72.0%
三维多项式	0.055	0.01	91.52%	93.0%
四维多项式	30	1	66.35%	71.0%

可以发现，在该数据集中，奇数维度的多项式核函数 SVM 分类效果姚明显优于偶数维度的多项式核函数 SVM，但它们的效果都不如使用高斯核函数的 SVM 模型。

5. 垃圾邮件分类的优化：

之前在“垃圾邮件分类”部分看到，对给定的两封垃圾邮件样例进行预测，发现对于 spamSample2.txt 的预测结果为 y=1，但对于 spamSample1.txt 的预测结果为 y=0，这说明该 SVM 也许还有一些改进的空间。现在加上两份样例邮件对比，通过调整惩罚系数 C 来得到优化该模型。

设置 C 值为[0.01, 0.03, 0.05, 0.1, 0.3, 1]运行结果如下：

```
When C is 0.05, accuracy rate for training set: 99.625
When C is 0.05, accuracy rate for testing set: 99.2
The soft of spamSample1: [0]
The soft of spamSample2: [1]
The soft of emailSample1: [0]
The soft of emailSample2: [0]


When C is 0.1, accuracy rate for training set: 99.825
When C is 0.1, accuracy rate for testing set: 98.9
The soft of spamSample1: [0]
The soft of spamSample2: [1]
The soft of emailSample1: [0]
The soft of emailSample2: [0]


When C is 0.3, accuracy rate for training set: 99.97500000000001
When C is 0.3, accuracy rate for testing set: 98.3
The soft of spamSample1: [1]
The soft of spamSample2: [1]
The soft of emailSample1: [0]
The soft of emailSample2: [0]


When C is 1, accuracy rate for training set: 99.97500000000001
When C is 1, accuracy rate for testing set: 97.8
The soft of spamSample1: [1]
The soft of spamSample2: [1]
The soft of emailSample1: [1]
The soft of emailSample2: [0]
```

将数据整理成表格，可以得到：

C	样例邮件 分类结果	垃圾邮件 分类结果	训练集准确率	验证集准确率
0.01	[0,0]	[0,0]	98.32%	98.0%
0.03	[0,0]	[0,1]	99.43%	99.0%
0.05	[0,0]	[0,1]	99.63%	99.2%
0.1	[0,0]	[0,1]	99.83%	98.9%
0.3	[0,0]	[1,1]	99.98%	98.3%
1	[1,0]	[1,1]	99.98%	97.8%

我们可以看到，当惩罚系数 C 不断增加，训练集的准确率会不断上升，这是非常符合预期，但是验证集的准确率会先上升，在 C=0.05 附近抵达峰值，之后就会逐渐下降，这说明模型会出现过拟合趋向，尤其是当 C 从 0.3 增大到 1 时，可以看到训练集的准确率不再变化，但是验证集准确率还在持续下降。

关于样例邮件和垃圾邮件分类情况，我们看到 C 值越大越可能将邮件判别为垃圾邮件。在所有 C 的取值中，只有 C=0.3 能够全部正确地进行分类，但是这时候模型已经有过拟合倾向了，这也反映出 spamSample1.txt 不是一个比较典型的垃圾邮件，想要分辨出它会比较困难。如果要追求更好的泛化性能，还是建议 C 取 0.03-0.1 之间的值。

四、实验心得

实验过程中遇到的什么问题。尝试使用什么方法去解决。通过实验获得了什么感悟与理解。

通过本次实验，我对支持向量机（SVM）的原理与应用有了更深刻的理解，同时在实践过程中积累了宝贵的经验。

在实验过程中，我遇到了不少的问题，在解决的过程中也锻炼了我的分析解决问题的能力。比如在进一步调整 SVM 的惩罚参数 C 与高斯核参数 σ 时，我刚开始想设置更多参考参数来实现，但发现这样很低效不说，得到的精度也很难保障。后来我尝试使用二分法来迭代参数的取值区间，但面对两个要优化的参数，我十分不知所措，刚开始选择同时及优化两个参数，发现非常没有逻辑连贯性，优化也老是锁死，直到后来借鉴了 BCD 的思想，交替优化不同的参数，该问题才得到了比较好的解决。

而在垃圾邮件分类任务中，我对惩罚系数 C 的理解又有了更新——当 C 值从 0.1 提升至 1 时，训练集准确率接近完美（99.98%），但验证集准确率却下降至 97.8%。这一现象表明模型过度拟合了训练数据中的噪声（如特定拼写变体的垃圾邮件），牺牲了泛化能力。通过对比不同 C 值下的分类结果（如 C=0.05 时验证集准确率达 99.2%），明确了惩罚参数需在拟合能力与泛化性之间权衡。这一教训启示，模型评估需始终以验证集性能为核心指标。