# 武汉大学国家网络安全学院

# 课程实践报告

# 机器学习实验

专 业 名 称 ：网络空间安全

课 程 名 称 ：机器学习

指 导 教 师 ：

学 生 学 号 ：

学 生 姓 名 ：

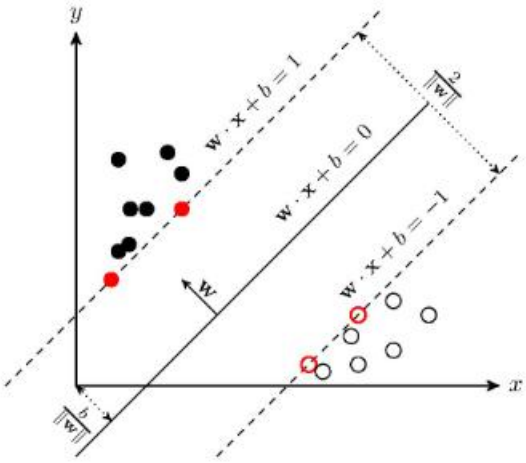
二〇二三年六月

**1☑实验内容及原理**（黑体 4 号）

一、SVM

支持向量机是一种二分类模型，它的基本模型是定义在特征空间上的间隔最大的线性分类器，间隔最大使它有别于感知机；SVM 还包括核技巧，这使它成为实质上的非线性分类器。

SVM 的的学习策略就是间隔最大化，可形式化为一个求解凸二次规划的问题，也等价于正则化的合页损失函数的最小化问题。SVM 的的学习算法就是求解凸二次规划的最优化算法。SVM 学习的基本想法是求解能够正确划分训练数据集并且几何间隔最大的分离超平面。如下图所示，wx+b=0　即为分离超平面，对于线性可分的数据集来说，这样的超平面有无穷多个（即感知机），但是几何间隔最大的分离超平面却是唯一的。




假设给定一个特征空间上的训练数据集

$$T=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_2,y_2),\ldots,(\boldsymbol{x}_N,y_N)\}$$

其中，$\boldsymbol{x}_i\in\mathbb{R}^n$ ，　$y_i\in\{+1,-1\},i=1,2,\ldots N$ ，xi 为第 i 个特征向量，yi 为类标记，当它等于+1 时为正例；为-1 时为负例。再假设训练数据集是线性可分的。

几何间隔：

$$\gamma_i=y_i\left(\frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}\cdot\boldsymbol{x_i}+\frac{b}{\|\boldsymbol{w}\|}\right)$$

超平面关于所有样本点的几何间隔的最小值为 $\gamma=\min\limits_{i=1,2\ldots,N}\gamma_i$

实际上这个距离就是我们所谓的支持向量到超平面的距离。根据以上定义，SVM 模型的求解最大分割超平面问题可以表示为以下约束最优化问题

$$\min_{\boldsymbol{w},b}\ \frac{1}{2}\|\boldsymbol{w}\|^2$$

$$s.t.\ \ y_i\left(\boldsymbol{w}\cdot\boldsymbol{x_i}+b\right)\geq 1,\ i=1,2,\ldots,N$$

对于任意训练样本（xi，yi），总有 ai=0 或者 yj（wxi+b）=1。若 ai=0，则该样本不会在最后求解模型参数的式子中出现，若 ai>0，则必有 yj（wxj+b）=1，所对应的样本点位于最大间隔边界上，是一个支持向量。这显示出支持向量机的一个重要性质：训练完成后，大部分的训练样本都不需要保留，最终模型仅与支持向量有关。

最终支持向量机学习算法如下：

**输入**：训练数据集 $T=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_1,y_1),\ldots,(\boldsymbol{x}_N,y_N)\}$ 其中，$\boldsymbol{x}_i\in\mathbb{R}^n$，$y_i\in\{+1,-1\},i=1,2,\ldots N$；

输出：分离超平面和分类决策函数

1. 选择乘法参数 C>0，构造并求解凸二次规划问题

$$\min_{\boldsymbol{\alpha}}\ \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j\left(\boldsymbol{x_i}\cdot\boldsymbol{x_j}\right)-\sum_{i=1}^{N}\alpha_i$$

$$s.t.\quad \sum_{i=1}^{N}\alpha_i y_i=0$$

$$0\le\alpha_i\le C,\ i=1,2,\ldots,N$$

得到最优解 $\boldsymbol{\alpha}^*=\left(\alpha_1^*,\alpha_2^*,\ldots,\alpha_N^*\right)^T$

2. 计算

$$\boldsymbol{w}^*=\sum_{i=1}^{N}\alpha_i^* y_i\boldsymbol{x}_i$$

选择 a* 的一个分量 aj* 满足条件 0<aj*<C，计算

$$b^*=y_j-\sum_{i=1}^{N}\alpha_i^* y_i\left(\boldsymbol{x_i}\cdot\boldsymbol{x_j}\right)$$

3. 求分离超平面

$$\boldsymbol{w}^*\cdot\boldsymbol{x}+b^*=0$$

分类决策函数 $f(\boldsymbol{x})=sign\left(\boldsymbol{w}^*\cdot\boldsymbol{x}+b^*\right)$

使用拉格朗日乘子法可得到其对偶问题，其拉格朗日函数可写为：

$$L(\omega, b, \alpha) = \frac{1}{2}\|\omega\|^2 + \sum_{i=1}^{m} \alpha_i (1 - y_i(\omega^T x_i + b))$$

利用对偶性的结论， 对 L 关于 ω 和 b 求偏导数

$$\frac{\partial L}{\partial \omega} = 0 \Rightarrow \omega = \sum_{i=1}^{n} \alpha_i x_i y_i$$

$$\frac{\partial L}{\partial b} = 0 \Rightarrow \sum_{i=1}^{n} \alpha_i y_i = 0$$

代入原式可得对偶问题

$$\max_{\alpha} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

$$s.t. \sum_{i=1}^{m} \alpha_i y_i = 0,\ \alpha_i \geq 0, i = 1, 2, ..., m$$

Sklearn 库中的 SVC 模型：

```
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)
```

C : float, optional (default=1.0)

误差项的惩罚参数，一般取值为 10 的 n 次幂，在 python 中可以使用 pow（10,n） n=-5~inf C 越大，相当于惩罚松弛变量，希望松弛变量接近 0，即对误分类的惩罚增大，趋向于对训练集全分对的情况，这样会出现训练集测试时准确率很高，但泛化能力弱。C 值小，对误分

类的惩罚减小，容错能力增强，泛化能力较强。

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

svc 中指定的 kernel 类型。可以是： 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' 或者自己指定。 默认使用'rbf' 。

degree : int, optional (default=3)

当指定 kernel 为 'poly'时，表示选择的多项式的最高次数，默认为三次多项式。若指定 kernel 不是'poly',则忽略，即该参数只对'poly'有作用。

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

当 kernel 为'rbf', 'poly'或'sigmoid'时的 kernel 系数。如果不设置，默认为 'auto' ，此时，kernel 系数设置为：1/n_features。

coef0 : float, optional (default=0.0)

kernel 函数的常数项。只有在 kernel 为'poly'或'sigmoid'时有效，默认为 0。

probability : boolean, optional (default=False)

是否采用概率估计。必须在 fit（）方法前使用，该方法的使用会降低运算速度，默认为 False。

shrinking : boolean, optional (default=True)

如果能预知哪些变量对应着支持向量，则只要在这些样本上训练就够了，其他样本可不予考虑，这不影响训练结果，但降低了问题的规模并有助于迅速求解。进一步，如果能预知哪些变量在边界上(即 a=C)，则这些变量可保持不动，只对其他变量进行优化，从而使问题的规模更小，训练时间大大降低。这就是 Shrinking 技术。Shrinking 技术基于这样一个事实：支持向量只占训练样本的少部分，并且大多数支持向量的拉格朗日乘子等于 C。

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

误差项达到指定值时则停止训练，默认为 1e-3，即 0.001。

cache_size : float, optional

指定内核缓存的大小，默认为 200M。

class_weight : {dict, 'balanced'}, optional

权重设置。如果不设置，则默认所有类权重值相同。以字典形式传入。

verbose : bool, default: False

是否启用详细输出。多线程时可能不会如预期的那样工作。默认为 False。

max_iter : int, optional (default=-1)

强制设置最大迭代次数。默认设置为-1，表示无穷大迭代次数。

decision_function_shape : ‘ovo’, ‘ovr’, default=’ovr’

决定了分类时，是一对多的方式来构建超平面，还是一对一。

random_state : int, RandomState instance or None, optional (default=None)

伪随机数使用数据。

二、KNeighborsClassifier

k 近邻法（k-nearest neighbor, kNN）是一种基本的分类与回归方法；是一种基于有标签训练数据的模型；是一种监督学习算法。

基本做法的三个要点是：第一，确定距离度量；第二，k 值的选择（找出训练集中与带估计点最靠近的 k 个实例点）；第三，分类决策规则。

在分类任务中可使用“投票法”，即选择这 k 个实例中出现最多的标记类别作为预测结果；在回归任务中可使用“平均法”，即将这 k 个实例的实值输出标记的平均值作为预测结果；还可基于距离远近进行加权平均或加权投票，距离越近的实例权重越大。k 近邻法不具有显式的学习过程。它是懒惰学习的著名代表，此类学习技术在训练阶段仅仅是把样本保存起来，训练时间开销为零，待收到测试样本后再进行处理。

特征空间中的两个实例点的距离是两个实例点相似程度的反映。K 近邻法的特征空间一般是 n 维实数向量空间 Rn。度量的距离是其他 Lp 范式距离，一般为欧式距离。

$$L_p(x_i, x_j) = \left(\sum_{l}^{n} |x_i^{(l)} + x_j^{(l)}|^p\right)^{\frac{1}{p}}$$

$$L_1(x_i, x_j) = \sum_{l}^{n} |x_i^{(l)} + x_j^{(l)}|$$

当 p=1 时，称为曼哈顿距离（Manhattan distance）

$$L_2(x_i, x_j) = \left(\sum_{l}^{n} |x_i^{(l)} + x_j^{(l)}|^2\right)^{\frac{1}{2}}$$

当 p=2 时，称为欧氏距离（Euclidean distance）

当 p=∞时，它是各个坐标距离的最大值。

k 值的选择:

（1）从 k=1 开始，使用检验集估计分类器的误差率。

（2）重复该过程，每次 K 增值 1，允许增加一个近邻。

（3）选取产生最小误差率的 K。

k 近邻法的实现：kd 树

k 近邻法最简单的实现方法是线性扫描，这时要计算输入实例与每一个训练实例的距离，当训练集很大时，计算非常耗时。

kNN 的时间复杂度为 O(n)，一般适用于样本数较少的数据集，当数据量大时，可以将数据以树的形式呈现，能提高速度，常用的有 kd-tree 和 ball-tree。

kd 树方法（平衡二叉树）：

kd 树是存储 k 维空间数据的树结构，这里的 k 与 k 近邻法的 k 意义不同。

依次选择坐标轴对空间切分，选择训练实例点在选定坐标轴上的中位数（一组数据按大小顺序排列起来，处于中间的一个数或最中间两个数的平均值。本文在最中间有两个数时选择最大值作中位数）为切分点，这样得到的 kd 树是平衡的。注意，平衡的 kd 树搜索时未必是最优的。

Sklearn 方法参数：

```
class sklearn.neighbors.KNeighborsClassifier(n_neighbors=5, weights='uniform',
                                              algorithm='auto', leaf_size=30,
                                              p=2, metric='minkowski',
                                              metric_params=None,
                                              n_jobs=None, **kwargs)
```

n_neighbors ： int，optional(default = 5)

默认情况下 kneighbors 查询使用的邻居数。就是 k-NN 的 k 的值，选取最近的 k 个点。

weights ： str 或 callable，可选(默认='uniform')

默认是 uniform，参数可以是 uniform、distance，也可以是用户自己定义的函数。uniform 是均等的权重，就说所有的邻近点的权重都是相等的。distance 是不均等的权重，距离近的点比距离远的点的影响大。用户自定义的函数，接收距离的数组，返回一组维数相同的权重。

algorithm ： {'auto'，'ball_tree'，'kd_tree'，'brute'}，可选

快速 k 近邻搜索算法，默认参数为 auto，可以理解为算法自己决定合适的搜索算法。除此之外，用户也可以自己指定搜索算法 ball_tree、kd_tree、brute 方法进行搜索，brute 是蛮力搜索，也就是线性扫描，当训练集很大时，计算非常耗时。kd_tree，构造 kd 树存储

数据以便对其进行快速检索的树形数据结构，kd 树也就是数据结构中的二叉树。以中值切分构造的树，每个结点是一个超矩形，在维数小于 20 时效率高。ball tree 是为了克服 kd 树高纬失效而发明的，其构造过程是以质心 C 和半径 r 分割样本空间，每个节点是一个超球体。

leaf_size ： int，optional(默认值= 30)

默认是 30，这个是构造的 kd 树和 ball 树的大小。这个值的设置会影响树构建的速度和搜索速度，同样也影响着存储树所需的内存大小。需要根据问题的性质选择最优的大小。

p ： 整数，可选(默认= 2)

距离度量公式。在上小结，我们使用欧氏距离公式进行距离度量。除此之外，还有其他的度量方法，例如曼哈顿距离。这个参数默认为 2，也就是默认使用欧式距离公式进行距离度量。也可以设置为 1，使用曼哈顿距离公式进行距离度量。

metric ： 字符串或可调用，默认为’minkowski’

用于距离度量，默认度量是 minkowski，也就是 p=2 的欧氏距离(欧几里德度量)。

metric_params ： dict，optional(默认=None)

距离公式的其他关键参数，这个可以不管，使用默认的 None 即可。

n_jobs ： int 或 None，可选(默认=None)

并行处理设置。默认为 1，临近点搜索并行工作数。如果为-1，那么 CPU 的所有 cores 都用于并行工作。

三、Id3

决策树是基于树状结构来进行决策的，一般地，一棵决策树包含一个根节点、若干个内部节点和若干个叶节点。每个内部节点表示一个属性上的判断，每个分支代表一个判断结果的输出，每个叶节点代表一种分类结果，根节点包含样本全集。

决策树学习的目的是为了产生一棵泛化能力强，即处理未见示例能力强的决策树，其基本流程遵循简单且直观的“分而治之”策略。

特征选择也即选择最优划分属性，从当前数据的特征中选择一个特征作为当前节点的划分标准。 随着划分过程不断进行，希望决策树的分支节点所包含的样本尽可能属于同一类别，即节点的“纯度”越来越高。

熵表示事务不确定性的程度，也就是信息量的大小（一般说信息量大，就是指这个时候背后的不确定因素太多），熵的公式如下：

$$Entropy = -\sum_{i=1}^{n} p(x_i) * log_2 p(x_i)$$

对于在 X 的条件下 Y 的条件熵，是指在 X 的信息之后，Y 这个变量的信息量（不确定性）的大小，计算公式如下：

$$Entropy(Y|X) = \sum_{i=1}^{n} p(x_i) Entropy(Y|x_i)$$

所以当 Entropy 最大为 1 的时候，是分类效果最差的状态，当它最小为 0 的时候，是完全分类的状态。因为熵等于零是理想状态，一般实际情况下，熵介于 0 和 1 之间 。

熵的不断最小化，实际上就是提高分类正确率的过程。

信息增益：在划分数据集之前之后信息发生的变化，计算每个特征值划分数据集获得的信息增益，获得信息增益最高的特征就是最好的选择。

定义属性 A 对数据集 D 的信息增益为 infoGain(D|A)，它等于 D 本身的熵，减去 给定 A 的条件下 D 的条件熵，即：

$$infoGain(D|A) = Entropy(D) - Entropy(D|A)$$

其中 $A = [a_1, a_2, \ldots, a_k]$ ， K个值。

信息增益的意义：引入属性 A 后，原来数据集 D 的不确定性减少了多少。

计算每个属性引入后的信息增益，选择给 D 带来的信息增益最大的属性，即为最优划分属性。

一般，信息增益越大，则意味着使用属性 A 来进行划分所得到的的“纯度提升”越大。

## 2☑实验步骤与分析（黑体 4 号）

一、数据处理

在实验中直接使用处理好的 csv 格式的数据集进行训练

```
def loadData(fileName):
    dataArr = []; labelArr = []
    fr = open(fileName)
    for line in fr.readlines():
        curLine = line.strip().split(',')
        #在放入的同时将原先字符串形式的数据转换为0-1的浮点型
        dataArr.append([int(num) / 255 for num in curLine[1:]])
        if int(curLine[0]) == 0:
            labelArr.append(1)
        else:
            labelArr.append(-1)
    return dataArr, labelArr
```

按行来读取，每行第一个是标签，去除多余的逗号，并将数据转换为浮点格式减少训练时间。

```
def ID3loadData(filename):
    # 存放数据以及标记
    dataArr = []
    labelArr = []
    # 读取文件
    fr = open(filename)
    # 遍历文件
    for line in fr.readlines():
        # strip:去除首尾部分的空格和回车
        curline = line.strip().split(",")
        # 数据二值化操作，减小运算量
        dataArr.append([int(int(num) > 128) for num in curline[1:]])
        # 添加标记
        labelArr.append(int(curline[0]))
        # 返回数据集和标记
    return dataArr, labelArr
```

在 ID3 算法中，因需要大量时间，故将数据特殊处理，将所有的数据转为 0 或 1 来减少训练时间。

二、SVM

```
clf=[SVC(),KNeighborsClassifier(),MySVM(),ID3()]
for i in range(2):
        start = time.time()
        clf[i].fit(trainData,trainLabel)
        end=time.time()
        print('accurate is:',clf[i].score(testData, testLabel),'time:',end-start)
```

通过调用 sklearn.svm 来直接使用 SVC（）模型。

三、KNeighborsClassifiers

同上，直接调用 sklearn.neighbors 直接使用

四、手动实现 SVM

```
class MySVM:

    def __init__(self,sigma = 10, C = 200, toler = 0.001):
        '''
        :param sigma: 高斯核中分母的σ
        :param C:软间隔中的惩罚参数
        :param toler:松弛变量
        '''
        self.trainDataMat = None          #训练数据集
        self.trainLabelMat = None         #训练标签集，为了方便后续运算提前做了转置，变为列向量
        self.m = None
        self.n = None                     #m：训练集数量     n：样本特征数目
        self.sigma = sigma                                  #高斯核分母中的σ
        self.C = C                                          #惩罚参数
        self.toler = toler                                  #松弛变量
        self.k = None                        #核函数（初始化时提前计算）
        self.b = 0                                          #SVM中的偏置b
        self.alpha = None    # α 长度为训练集数目
        self.E = None     #SMO运算过程中的Ei
        self.supportVecIndex = []                           #支持向量集
```

首先定义一个类，并初始化各项参数。

```
def calcKernel(self):
    '''
    :return: 高斯核矩阵
    '''
    #初始化高斯核结果矩阵 大小 = 训练集长度m * 训练集长度m
    #k[i][j] = Xi * Xj
    k = [[0 for i in range(self.m)] for j in range(self.m)]

    #大循环遍历Xi，Xi为式7.90中的x
    for i in range(self.m):
        X = self.trainDataMat[i, :]
        #小循环遍历Xj，Xj为式7.90中的Z
        # 由于 Xi * Xj 等于 Xj * Xi，一次计算得到的结果可以
        # 同时放在k[i][j]和k[j][i]中，这样一个矩阵只需要计算一半即可
        #所以小循环直接从i开始
        for j in range(i, self.m):
            #获得Z
            Z = self.trainDataMat[j, :]
            #先计算||X - Z||^2
            result = (X - Z) * (X - Z).T
            #分子除以分母后去指数，得到的即为高斯核结果
            result = np.exp(-1 * result / (2 * self.sigma**2))
            #将Xi*Xj的结果存放入k[i][j]和k[j][i]中
            k[i][j] = result
            k[j][i] = result
    #返回高斯核矩阵
    return k
```

定义核函数

```
def isSatisfyKKT(self, i):
    '''
    查看第i个α是否满足KKT条件
    :param i:α的下标
    '''
    gxi =self.calc_gxi(i)
    yi = self.trainLabelMat[i]
    if (math.fabs(self.alpha[i]) < self.toler) and (yi * gxi >= 1):
        return True
    elif (math.fabs(self.alpha[i] - self.C) < self.toler) and (yi * gxi <= 1):
        return True
    elif (self.alpha[i] > -self.toler) and (self.alpha[i] < (self.C + self.toler)) \
            and (math.fabs(yi * gxi - 1) < self.toler):
        return True

    return False
```

判断第 i 个拉格朗日乘子是否满足条件范围

```
def calc_gxi(self, i):
    '''
    计算g(xi)
    :param i:x的下标
    :return: g(xi)的值
    '''
    gxi = 0
    #index获得非零α的下标，并做成列表形式方便后续遍历
    index = [i for i, alpha in enumerate(self.alpha) if alpha != 0]
    #遍历每一个非零α，i为非零α的下标
    for j in index:
        #计算g(xi)
        gxi += self.alpha[j] * self.trainLabelMat[j] * self.k[j][i]
    #求和结束后再单独加上偏置b
    gxi += self.b
    #返回
    return gxi
```

只需要选择支持向量，因为只有支持向量才会对边界产生影响。

```
def calcEi(self, i):
    '''
    计算Ei
    :param i: E的下标
    :return:
    '''
    #计算g(xi)
    gxi = self.calc_gxi(i)
    #Ei = g(xi) - yi,直接将结果作为Ei返回
    return gxi - self.trainLabelMat[i]
```

```python
def getAlphaJ(self, E1, i):
    '''
    SMO中选择第二个变量
    :param E1: 第一个变量的E1
    :param i: 第一个变量α的下标
    :return: E2, α2的下标
    '''
    #初始化E2
    E2 = 0
    #初始化|E1-E2|为-1
    maxE1_E2 = -1
    #初始化第二个变量的下标
    maxIndex = -1
    #获得Ei非0的对应索引组成的列表，列表内容为非0Ei的下标i
    nozeroE = [i for i, Ei in enumerate(self.E) if Ei != 0]
    #对每个非零Ei的下标i进行遍历
    for j in nozeroE:
        #计算E2
        E2_tmp = self.calcEi(j)
        #如果|E1-E2|大于目前最大值
        if math.fabs(E1 - E2_tmp) > maxE1_E2:
            #更新最大值
            maxE1_E2 = math.fabs(E1 - E2_tmp)
            #更新最大值E2
            E2 = E2_tmp
            #更新最大值E2的索引j
            maxIndex = j
    #如果列表中没有非0元素了（对应程序最开始运行时的情况）
    if maxIndex == -1:
        maxIndex = i
        while maxIndex == i:
            #获得随机数，如果随机数与第一个变量的下标i一致则重新随机
            maxIndex = int(random.uniform(0, self.m))
        #获得E2
        E2 = self.calcEi(maxIndex)
    #返回第二个变量的E2值以及其索引
    return E2, maxIndex
```

进行随机梯度下降。

```python
    def fit(self,trainData,trainLabel, iter = 100):
        self.trainDataMat = np.mat(trainData)       #训练数据集
        self.trainLabelMat = np.mat(trainLabel).T   #训练标签集，为了方便后续运算提前做了转置，变为列向量
        self.m, self.n = np.shape(self.trainDataMat)
        self.alpha = [0] * self.trainDataMat.shape[0]
        self.E = [0 * self.trainLabelMat[i, 0] for i in range(self.trainLabelMat.shape[0])]
        self.k = self.calcKernel()
```

```
        #iterStep：迭代次数，超过设置次数还未收敛则强制停止
        #parameterChanged：单次迭代中有参数改变则增加1
        iterStep = 0; parameterChanged = 1
        #如果没有达到限制的迭代次数以及上次迭代中有参数改变则继续迭代
        #parameterChanged==0时表示上次迭代没有参数改变，如果遍历了一遍都没有参数改变，说明
        #达到了收敛状态，可以停止了
        while (iterStep < iter) and (parameterChanged > 0):
            #打印当前迭代轮数
            #print('iter:%d:%d'%( iterStep, iter))
            #迭代步数加1
            iterStep += 1
            #新的一轮将参数改变标志位重新置0
            parameterChanged = 0
            #大循环遍历所有样本，用于找SMO中第一个变量
            for i in range(self.m):
                #查看第一个遍历是否满足KKT条件，如果不满足则作为SMO中第一个变量从而进行优化
                if self.isSatisfyKKT(i) == False:
                    #如果下标为i的α不满足KKT条件，则进行优化
                    #第一个变量α的下标i已经确定，接下来按照“7.4.2 变量的选择方法”第二步
                    #选择变量2。由于变量2的选择中涉及到|E1 - E2|，因此先计算E1
                    E1 = self.calcEi(i)
                    #选择第2个变量
                    E2, j = self.getAlphaJ(E1, i)
                    #参考“7.4.1 两个变量二次规划的求解方法” P126 下半部分
                    #获得两个变量的标签
                    y1 = self.trainLabelMat[i]
                    y2 = self.trainLabelMat[j]
                    #复制α值作为old值
                    alphaOld_1 = self.alpha[i]
                    alphaOld_2 = self.alpha[j]
                    #依据标签是否一致来生成不同的L和H
                    if y1 != y2:
                        L = max(0, alphaOld_2 - alphaOld_1)
                        H = min(self.C, self.C + alphaOld_2 - alphaOld_1)
                    else:
                        L = max(0, alphaOld_2 + alphaOld_1 - self.C)
                        H = min(self.C, alphaOld_2 + alphaOld_1)
                    #如果两者相等，说明该变量无法再优化，直接跳到下一次循环
                    if L == H:   continue
                    #计算α的新值
                    #先获得几个k值，用来计算事7.106中的分母η
                    k11 = self.k[i][i]
                    k22 = self.k[j][j]
                    k21 = self.k[j][i]
```

```
                    k12 = self.k[i][j]
                    #依据式7.106更新α2，该α2还未经剪切
                    alphaNew_2 = alphaOld_2 + y2 * (E1 - E2) / (k11 + k22 - 2 * k12)
                    #剪切α2
                    if alphaNew_2 < L: alphaNew_2 = L
                    elif alphaNew_2 > H: alphaNew_2 = H
                    #更新α1，依据式7.109
                    alphaNew_1 = alphaOld_1 + y1 * y2 * (alphaOld_2 - alphaNew_2)
                    #依据“7.4.2 变量的选择方法”第三步式7.115和7.116计算b1和b2
                    b1New = -1 * E1 - y1 * k11 * (alphaNew_1 - alphaOld_1) \
                            - y2 * k21 * (alphaNew_2 - alphaOld_2) + self.b
                    b2New = -1 * E2 - y1 * k12 * (alphaNew_1 - alphaOld_1) \
                            - y2 * k22 * (alphaNew_2 - alphaOld_2) + self.b
                    #依据α1和α2的值范围确定新b
                    if (alphaNew_1 > 0) and (alphaNew_1 < self.C):
                        bNew = b1New
                    elif (alphaNew_2 > 0) and (alphaNew_2 < self.C):
                        bNew = b2New
                    else:
                        bNew = (b1New + b2New) / 2
                    #将更新后的各类值写入，进行更新
                    self.alpha[i] = alphaNew_1
                    self.alpha[j] = alphaNew_2
                    self.b = bNew
                    self.E[i] = self.calcEi(i)
                    self.E[j] = self.calcEi(j)
                    #如果α2的改变量过于小，就认为该参数未改变，不增加parameterChanged值
                    if math.fabs(alphaNew_2 - alphaOld_2) >= 0.00001:
                        parameterChanged += 1
                #打印迭代轮数，i值，该迭代轮数修改α数目
                #print("iter: %d i:%d, pairs changed %d" % (iterStep, i, parameterChanged))
        #全部计算结束后，重新遍历一遍α，查找里面的支持向量
        for i in range(self.m):
            #如果α>0，说明是支持向量
            if self.alpha[i] > 0:
                #将支持向量的索引保存起来
                self.supportVecIndex.append(i)
```

进行训练，开始时接收传入的参数和训练数据。

```
def calcSinglKernel(self, x1, x2):
    '''
    单独计算核函数
    :param x1:向量1
    :param x2: 向量2
    :return: 核函数结果
    '''
    result = (x1 - x2) * (x1 - x2).T
    result = np.exp(-1 * result / (2 * self.sigma ** 2))
    #返回结果
    return np.exp(result)
```

核函数计算，这里使用线性核。

```
def predict(self, x):
    result = 0
    for i in self.supportVecIndex:
        tmp = self.calcSinglKernel(self.trainDataMat[i, :], np.mat(x))
        result += self.alpha[i] * self.trainLabelMat[i] * tmp
    result += self.b
    return np.sign(result)

def score(self, testDataList, testLabelList):
    errorCnt = 0
    for i in range(len(testDataList)):
        #print('test:%d:%d'%(i, len(testDataList)))
        result = self.predict(testDataList[i])
        if result != testLabelList[i]:
            errorCnt += 1
    return 1 - errorCnt / len(testDataList)
```

预测与准确率计算。

五、ID3

```
def CreateTree(self,*dataSet):
    '''
递归创建决策树
:param dataSet:(trainDataList, trainLabelList) <<-- 元祖形式
:return:新的子节点或该叶子节点的值
'''
    Epsilon = 0.1

    trainDataList = dataSet[0][0]
    trainLabelList = dataSet[0][1]

# 创建子节点，打印当前的特征个数，以及当前的样本个数
    #print("start a new lode,当前的特征个数为:%d，样本个数为:%d" % (len(trainDataList[0]), len(trainLabelList)))

# 将标签放入一个字典中，当前样本有多少类，在字典中就会有多少项
    classDict = {i for i in trainLabelList}
    if len(classDict) == 1:
    # 因为所有样本都是一致的，在标签集中随便拿一个标签返回都行
        return trainLabelList[0]

# 当特征个数为空时，返回占最多数的类别标签
    if len(trainDataList[0]) == 0:
        return self.MajorClass(trainLabelList)

# 否则，计算每个特征的信息增益，选择信息增益最大的特征Ag
    Ag, max_GDA = self.Cal_BestFeature(trainDataList, trainLabelList)

# Ag的信息增益小于阈值Epsilon，则置T为单节点树，并将D中实例数最大的类Ck，作为该节点的类，返回T。
    if max_GDA < Epsilon:
        return self.MajorClass(trainLabelList)

# 否则，对Ag的每一个可能值ai，依据Ag=ai将数据集分割为若干个非空子集Di，将Di中实例数最大的类作为标记，
# 构建子节点，由节点及其子节点构成树T，返回T。
    treeDict = {Ag: {}}
    treeDict[Ag][0] = self.CreateTree(self.GetSubDataArr(trainDataList, trainLabelList, Ag, 0))
    treeDict[Ag][1] = self.CreateTree(self.GetSubDataArr(trainDataList, trainLabelList, Ag, 1))
    return treeDict
```

```
def MajorClass(self,labelArr):
    '''
找到当前标签集中占数目最大的标签
:param labelArr: 标签集
:return: 最大的标签
    '''
# 建立字典，统计不同类别标签的数量
    classDict = {}
# 遍历所有标签
    for i in range(len(labelArr)):
        if labelArr[i] in classDict.keys():
            classDict[labelArr[i]] += 1
        else:
            classDict[labelArr[i]] = 1
# 对不同类别标签的统计情况进行降序排序
    classSort = sorted(classDict.items(), key=lambda x: x[1], reverse=True)
# 返回最大一项的标签，即占数目最多的标签
    return classSort[0][0]
```

```
def Cal_HD(self,trainLabelArr):
    '''
计算数据集D的经验熵，参考公式经验熵的计算
:param trainLabelArr:当前数据集的标签集
:return: 经验熵
'''
    HD = 0
# 统计该分支的标签情况
# set()删除重复数据
    trainLabelSet = set([label for label in trainLabelArr])
# 遍历每一个出现过的标签
    for i in trainLabelSet:
        p = trainLabelArr[trainLabelArr == i].size / trainLabelArr.size
    # 对经验熵的每一项进行累加求和
        HD += -1 * p * np.log2(p)

# 返回经验熵
    return HD
```

```
def Cal_HDA(self,trainDataArr_DevFeature, trainLabelArr):
    '''
计算经验条件熵
:param trainDataArr_DevFeature:切割后只有feature那列数据的数组
:param trainLabelArr: 标签集数组
:return: 条件经验熵
'''
# 初始为0
    HDA = 0
# 拿到当前指定feature中的可取值的范围
    trainDataSet = set([label for label in trainDataArr_DevFeature])

# 对于每一个特征取值遍历计算条件经验熵的每一项
# trainLabelArr[trainDataArr_DevFeature == i]表示特征值为i的样本集对应的标签集
    for i in trainDataSet:
        HDA += trainDataArr_DevFeature[trainDataArr_DevFeature == i].size / trainDataArr_DevFeature.size * self.Cal_HD(
            trainLabelArr[trainDataArr_DevFeature == i])
# 返回得出的条件经验熵
    return HDA
```

```
def Cal_BestFeature(self,trainDataList, trainLabelList):
    '''
计算信息增益最大的特征
:param trainDataList: 当前数据集
:param trainLabelList: 当前标签集
:return: 信息增益最大的特征及最大信息增益值
'''
# 列表转为数组格式
    trainDataArr = np.array(trainDataList)
    trainLabelArr = np.array(trainLabelList)

# 获取当前的特征个数
# 获取trainDataArr的列数
    featureNum = trainDataArr.shape[1]

# 初始化最大信息熵G(D|A)
    max_GDA = -1
# 初始化最大信息增益的特征索引
    max_Feature = -1

# 计算数据集的经验熵
    HD = self.Cal_HD(trainLabelArr)
# 对每一个特征进行遍历计算
    for feature in range(featureNum):
    # .flat：flat返回的是一个迭代器，可以用for访问数组每一个元素
        trainDataArr_DevideByFeature = np.array(trainDataArr[:, feature].flat)

    # 计算信息增益G(D|A) = H(D) - H(D|A)
        GDA = HD - self.Cal_HDA(trainDataArr_DevideByFeature, trainLabelArr)

    # 不断更新最大的信息增益以及对应的特征
        if GDA > max_GDA:
            max_GDA = GDA
            max_Feature = feature
    return max_Feature, max_GDA
```

```
def GetSubDataArr(self,trainDataArr, trainLabelArr, A, a):
    '''
待更新的子数据集和标签集
:param trainDataArr: 待更新的数据集
:param trainLabelArr: 待更新的标签集
:param A: 待去除的选定特征
:param a: 当data[A]==a时，该行样本保留
:return: 新的数据集和标签集
'''
# 返回的数据集，标签集
    retDataArr, retLabelArr = [], []

# 对当前数据集的每一个样本进行遍历
    for i in range(len(trainDataArr)):
    # 如果当前样本的特征为指定特征值a
        if trainDataArr[i][A] == a:
        # 那么将该样本的第A个特征切割掉，放入返回的数据集中
            retDataArr.append(trainDataArr[i][0:A] + trainDataArr[i][A + 1:])
        # 将该样本的标签放入新的标签集中
            retLabelArr.append(trainLabelArr[i])

# 返回新的数据集和标签集
    return retDataArr, retLabelArr

def Predict(self,testDataList,tree):

# 死循环，直到找到一个有效地分类
    while True:
    # 以tree={73: {0: {74:6}}}为例,key=73，value={0: {74:6}}
        (key, value), = tree.items()
    # 如果当前的value是字典，说明还需要遍历下去
        if type(tree[key]).__name__ == 'dict':
        # 获取目前所在节点的feature值，需要在样本中删除该feature
        # 因为在创建树的过程中，feature的索引值永远是对于当时剩余的feature来设置的
        # 所以需要不断地删除已经用掉的特征，保证索引相对位置的一致性
            dataVal = testDataList[key]
            del testDataList[key]
        # 将tree更新为其子节点的字典
            tree = value[dataVal]
            if type(tree).__name__ == 'int':
            # 如果当前节点的子节点的值是int，就直接返回该int值
            # 以{403: {0: 7, 1: {297:7}}为例，dataVal=0，则当前节点的子节点的值是7，为int型
            # 返回该节点值，也就是分类值
                return tree
        else:
        # 如果当前value不是字典，那就返回分类值
        # 以{297:7}为例，key=297,value=7，则直接返回7
            return value
```

```
def Model_Test(self,testDataList, testLabelList,tree):
    '''
测试准确率
:param testDataList:待测试数据集
:param testLabelList: 待测试标签集
:param tree: 训练集生成的树
:return: 准确率
'''
# 错误次数计数
    errorCnt = 0
# 遍历测试集中每一个测试样本
    for i in range(len(testDataList)):
    # 判断预测与标签中结果是否一致
        if testLabelList[i] != self.Predict(testDataList[i],tree):
            errorCnt += 1
# 返回准确率
    print('accurate is:',1 - errorCnt / len(testDataList),end=' ')
```

首先要创建出一棵决策树来，然后根据树来进行预测。

## 3☑实验结果与总结（黑体 4 号）

一、参数测试

1．SVC

| C | 准确率 |
|---|---|
| 1 | 0.9977 |
| 0.8 | 0.9976 |
| 0.5 | 0.9974 |
| 0.3 | 0.9973 |
| 0.1 | 0.9966 |

测试得出 C=1 时准确率最高。

2．KNN

| Neighbors | 准确率 |
|---|---|
| 1 | 0.9973 |
| 2 | 0.9976 |
| 3 | 0.997 |
| 4 | 0.997 |
| 5 | 0.9965 |
| 6 | 0.997 |
| 7 | 0.9965 |
| 8 | 0.9963 |

测试得出 neighbors=2 时准确率最高。

3．MySVM

| Sigma（default=10） | 准确率 | C（default=200） | 准确率 | Toler（default=0.001） | 准确率 |
|---|---|---|---|---|---|
| 10 | 0.968 | 100 | 0.968 | 0.001 | 0.961 |
| 20 | 0.966 | 200 | 0.967 | 0.01 | 0.932 |
| 50 | 0.942 | 500 | 0.935 | 0.1 | 0.971 |
| 100 | 0.934 | 1000 | 0.902 | 1 | 0.9921 |

经过多轮测试，准确率会波动。因测试时间过长，仅选取训练集与测试集前 1000 数据。

二、性能比较

```
start read TrainDataSet
start read TestDataSet
start to train, please wait...
accurate is: 0.9977 time: 51.081186294555664
accurate is: 0.9965 time: 1.9346506595611572
accurate is: 0.946 time: 1622.528856754303
accurate is: 0.8589 time: 126.73412346839905

start read TrainDataSet
start read TestDataSet
start to train, please wait...
accurate is: 0.9977 time: 41.45235300064087
accurate is: 0.9965 time: 1.9069607257843018
accurate is: 0.964 time: 873.0175888538361
accurate is: 0.8589 time: 137.52677989006042
```

多次测试后，可以看出手动实现的 SVM 算法时间耗时最长，而且训练集也只使用了原本的 10%，其次是 ID3，因为简化了数据集因此时间还在可接受范围内，KNN 最快，因为是懒惰学习没有时间代价。

三、算法比较

1．SVC 与 SVM

| SVC 准确率 | SVM 准确率 | 时间（SVM/SVC） |
|---|---|---|
| 0.9977 | 0.968 | 31.8 |
| 0.9977 | 0.968 | 31.2 |
| 0.9977 | 0.961 | 28.8 |
| 0.9977 | 0.967 | 28.6 |
| 0.9977 | 0.971 | 21.3 |

可以看出 SVM 的准确率还是不及 SVC，时间上更是难以企及。SVC 详细数据如下。

```
accurate is: 0.9977 time: 64.55291080474854
              precision    recall  f1-score   support

          -1       1.00      1.00      1.00      9020
           1       0.99      0.98      0.99       980

    accuracy                           1.00     10000
   macro avg       1.00      0.99      0.99     10000
weighted avg       1.00      1.00      1.00     10000
```

2．SVM 与 ID3

| ID3 准确率 | SVM 准确率 | 时间（SVM/ID3） |
|---|---|---|
| 0.8589 | 0.96 | 11.2 |
| 0.8589 | 0.961 | 11.1 |
| 0.8589 | 0.967 | 11 |
| 0.8589 | 0.968 | 11.5 |
| 0.8589 | 0.971 | 11.3 |

3．SVM 与 KNN

| KNN 准确率 | SVM 准确率 |
|---|---|
| 0.9965 | 0.96 |
| 0.9965 | 0.968 |
| 0.9965 | 0.961 |
| 0.9965 | 0.967 |
| 0.9965 | 0.971 |

KNN 速度太快已经没有什么时间比较的必要了，KNN 详细数据如下图。

```
accurate is: 0.9965 time: 2.3329601287841797
              precision    recall  f1-score   support

          -1       1.00      1.00      1.00      9020
           1       0.97      0.99      0.98       980

    accuracy                           1.00     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       1.00      1.00      1.00     10000
```

总结：总体来说 sklearn 库中的分类算法不仅准确率高而且运行速度快，在对于 mnist 数据集分类 SVC 准确率最高，然后是 KNN，手动实现的 SVM 运行时间实在太长在少量数据训练的情况下准确率不错，ID3 的准确率最低，时间开销也比较大。

## 教师评语评分

评语：_____

_____

_____

_____

评分：_____

评阅人：

年　　月　　日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）