第十三章 手撕机器学习

对数据的分析和处理离不开机器学习方法。本章对经典机器学习方法从原理、算法和代码进行了全方位的讲解，没有调用任何封装好的机器学习算法库，核心目的是为了从细节上详细了解每个算法的详细实现过程，从而对这些算法进行详尽的理解，同时深入掌握Numpy和Pandas等库函数的实用技巧。

本章重点难点：从代码角度对经典机器学习方法的理解。

13.1 感知机

13.1.1 感知机模型

感知机是线性的二分类模型，其输入为实例的特征向量，输出为实例的类别。感知机构造了一个分离超平面, 将输入空间(特征空间)分为正（+1）负（-1）两类的。超平面本质上是一个分割面, 它在二维上体现为直线，在三维上为平面，在高维上称为超平面。感知机学习就是通过训练数据确定能够将输入空间分为正负两部分的分离超平面。感知机学习算法具有简单而易于实现的优点，分为原始形式和对偶形式。感知机是神经网络与支持向量机的基础。感知机的函数如下：

$$f(x)=sign(\omega \cdot x+b)$$

其中$x \in R^n$表示实例的特征向量，R^n中R表示实数，指数n表示n维向量，即$x=(x^{(1)},x^{(2)},...x^{(n)})^T$，$x^{(i)} \in R$表示数据$x$的第$i$维特征；输出$y \in \{+1,-1\}$表示实例的类别；$\omega \in R^n$和$b \in R$表示感知机模型参数, 分别叫作权值和偏置。$\omega \cdot x$表示$\omega$和$x$的内积，$sign$是符号函数：

$$(x)=\{+1,\ x \geqslant 0\ -1,\ x<0$$

感知机模型是定义在特征空间中的所有线性分类模型，即函数集合$\{f \mid f(x)=\omega \cdot x+b\}$。

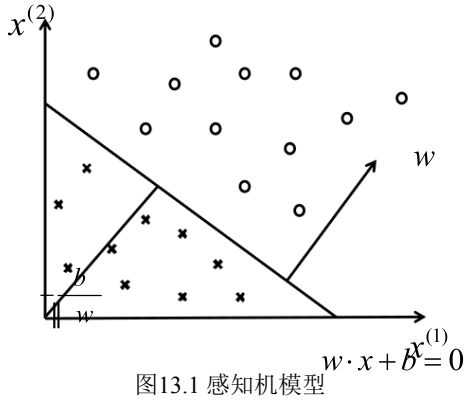


图13.1 感知机模型

对于误分类数据(x_i, y_i), 当$\omega \cdot x_i+b>0$时，$y_i=-1$；当$\omega \cdot x_i+b<0$时，$y=1$，即：

$$-y_i(\omega \cdot x_i+b)>0$$

因此, 感知机$sign(w \cdot x+b)$的损失函数定义为：

$$L(w,b) = -\sum_{x_i \in M} y_i\left(w \cdot x_i + b\right)$$

其中M为误分类点的集合。损失函数$L(\omega, b)$是非负的，表达了所有误分类数据偏离程度的总和。无误分类点时损失函数值为0，在误分类时损失函数是参数ω, b的线性函数。因此，损失函数$L(\omega, b)$是w, b的连续可导函数。误分类点越少，误分类点离超平面越近，损失函数就越小，其中$-y_i(\omega \cdot x_i + b)$表示误分类点$(x_i, y_i)$到超平面的相对距离。

13.1.2 感知机算法的原始形式

设给定训练数据集为：

$$\{(x_1, y_1), (x_2, y_2) \ldots, (x_N, y_N)\}$$

其中，$x_i \in R^n$, $y_i \in \{-1, 1\}$, $i = 1, 2, \ldots, N$, 求参数ω, b, 使其为以下损失函数极小化问题的解

$$L(w,b) = -\sum_{x_i \in M} y_i\left(w \cdot x_i + b\right)$$

其中M为误分类点的集合。

感知机算法用梯度下降法不断地极小化目标函数。首先，任意选取一个超平面ω_0, b_0, 然后不断地极小化目标函数。极小化过程中每次随机选取一个误分类点使其梯度下降。损失函数$L(\omega, b)$的梯度为：

$$\nabla_w L(w,b) = -\sum_{x_i \in M} y_i x_i \qquad \nabla_b L(w,b) = -\sum_{x_i \in M} y_i$$

随机选取一个误分类点(x_i, y_i), 对ω, b进行更新，下式中$\eta(0<\eta\leq 1)$是步长，也叫学习率。

$$\omega \leftarrow \omega + \eta y_i x_i \qquad b \leftarrow b + \eta y_i$$

算法**13.1.1 (**感知机算法的原始形式**)**

输入：训练数据集$(x_1, y_1), (x_2, y_2) \ldots, (x_N, y_N)\}$，其中$x_i \in R^n$, $y_i \in \{-1, +1\}$, $i=1, 2, .., N$; 学习率$\eta(0<\eta\leq 1)$;

输出：ω, b; 感知机模型$f(x) = sign(\omega \cdot x_i + b)$。

(1) 选取初值ω_0, b_0;

(2) 在训练集中选取数据(x_i, y_i);

(3) 如果$y_i(\omega \cdot x_i + b) \leq 0$,

$$\omega \leftarrow \omega + \eta y_i x_i \qquad b \leftarrow b + \eta y_i$$

(4) 转至(2)，直至训练集中没有误分类点。■

例题：编程实现感知机算法的原始形式，求模型$f(x) = sign(\omega \cdot x + b)$。

首先, 引入基本的库函数并进行画图参数配置。

```
01 import numpy as np
02 import pandas as pd
03 import matplotlib.pyplot as plt
04 #画图参数配置
05 plt.rcParams['font.family'] = ['sans-serif']
06 plt.rcParams['font.sans-serif'] = ['SimHei']
07 plt.rcParams['axes.unicode_minus']=False
08 plt.axes().set_aspect('equal')                                    # 使x,y轴的单位长度一致
```

根据感知机算法的原始形式, 构造具体实现过程。

```
01 def perceptron(data,w,b,lr):                                     # 数据, 参数, 偏置, 学习率
02     X=data[:,:-1]
03     Y=data[:,-1]
04     count=0                                                     # 迭代次数
05     ret = [[0,None,0,0,0]]
06     error_point = not None
07     while error_point:
08         hyperplane(count,X,Y,w,b)
09         error_point=None
10         for i in range(len(X)):
11             if Y[i]*(np.dot(w,X[i])+b)<=0:          # 误分类
12                 w=w+lr*Y[i]*X[i]                            # 更新参数
13                 b=b+lr*Y[i]                                 # 更新偏置
14                 error_point = f'x{i+1}'                     # 记录误分类点
15                 break
16         count+=1
17         if(error_point): ret.append([count,error_point,w,b,f'{w[0]}*x1+{w[1]}*x2+{b}'])
18     return  ret
```

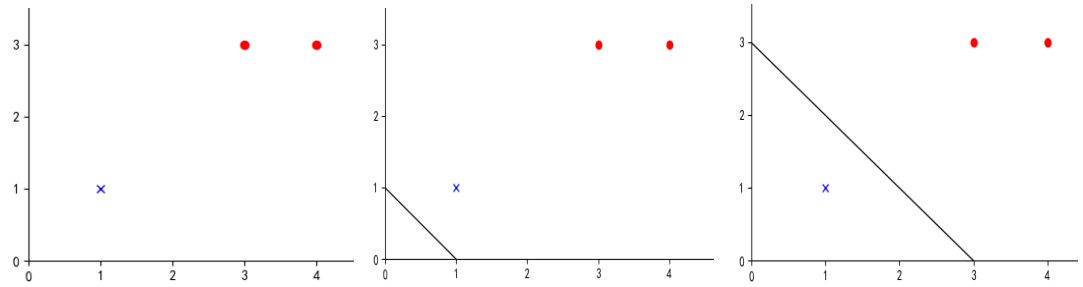
为了直观的展现超平面的效果，采用hyperplane函数进行绘制。

```
01 def hyperplane(index,X,Y,w,b):
02     positive = X[np.where(Y==1)]
         # 正样本
03     plt.plot(positive[:,0], positive[:,1],'ro')
         # 画正样本
04     negative = X[np.where(Y==-1)]
         # 负样本
05     plt.plot(negative[:,0], negative[:,1],'bx')
         # 画负样本
06     xlist = np.linspace(0.0, 5.0, 100)
07     y_line = [float('inf')]*len(xlist) if w[1]==0 else (-w[0]*xlist-b)/w[1]
08     plt.plot(xlist, y_line, color='k', linewidth=1, linestyle='-')          # 绘制超平面直线
09     plt.axis([0.0, 5.0, 0.0, 5.0]) # 坐标轴显示范围
10     plt.title(f'第{index}次迭代结果')
11     plt.show()
```

最后，读取数据，调用以上函数完成感知机算法的分类。

```
01 data=pd.read_csv('example1.1.csv',header=None,encoding='utf8').values
02 w,b,lr=np.array([0,0]),0,1   #参数初始化
03 ret=perceptron(data,w,b,lr)
04 df = pd.DataFrame(ret,columns=['迭代次数','误分类点','w','b','模型'])
05 print(df)
```

	迭代次数	误分类点	w	b	模型
0	0	None	0	0	0
1	1	x1	[3, 3]	1	3*x1+3*x2+1
2	2	x3	[2, 2]	0	2*x1+2*x2+0
3	3	x3	[1, 1]	-1	1*x1+1*x2+-1
4	4	x3	[0, 0]	-2	0*x1+0*x2+-2
5	5	x1	[3, 3]	-1	3*x1+3*x2+-1
6	6	x3	[2, 2]	-2	2*x1+2*x2+-2
7	7	x3	[1, 1]	-3	1*x1+1*x2+-3



原始数据　　第3次迭代　　第7次迭代

图13.2 超平面绘制结果(正实例点是$x_1 = (3,3)^T$, $x_2 = (4,3)^T$, 负实例点是$x_3 = (1,1)^T$)

13.1.3 感知机算法的对偶形式

根据感知机算法的原始形式，不失一般性，可以加上初始值ω_0, b_0均为0，因此ω, b用以下形式表达：

$$\omega = \sum_{i=1}^{N} \alpha_i y_i x_i \qquad b = \sum_{i=1}^{N} \alpha_i y_i \mathrm{b}$$

其中，$\alpha_i = n_i \eta \geq 0,\ i \in [1, N]$，而$n_i$表示第$i$个实例点由于误分而进行更新的次数，参数$\omega$和偏置$b$关于$(x_i, y_i)$的增量分别是$\alpha_i y_i x_i$和$\alpha_i y_i$。

算法**13.1.2 (**感知机算法的对偶形式**)**

输入：训练数据集$\{(x_1, y_1), (x_2, y_2) \ldots, (x_N, y_N)\}$，其中$x_i \in R^n$，$y_i \in \{-1, +1\}$，$i = 1, 2, .., N$；学习率$\eta(0 < \eta \leq 1)$;

输出：α, b; 感知机模型$f(x) = sign\left(\sum_{j=1}^{N} \alpha_j y_j x_j \cdot x + b\right)$，$\alpha = (\alpha_1, \alpha_2, \ldots \alpha_N)^T$

(1) $\alpha \leftarrow 0,\ b \leftarrow 0$;

(2) 在训练集中选取数据$(x_i,\ y_i)$;

(3) 如果$y_i\left(\sum_{j=1}^{N} \alpha_j y_j x_j \cdot x_i + b\right) \leq 0$,

$$\alpha_i \leftarrow \alpha_i + \eta \quad b \leftarrow b + \eta y_i$$

(4) 转至(2)，直至训练集中没有误分类点。■

在对偶形式的算法中，最重要的是训练实例以以内积的形式出现。因此可以将训练集中实例间的内积进行预计算并存储为矩阵，该矩阵就是所谓的Gram矩阵。

这种内积形式非常重要，它意味着可以使用核函数代替向量间的计算，从而解决非线性分类问题。对于非线性分类，可以通过一个映射函数将原本输入的非线性特征空间转化为线性特征空间，从而采用线性分类算法进行求解，例如感知机算法或支持向量机算法。但输入的特征空间一般是高维的，甚至是无穷维的，无论是特征的定义还是映射函数的选取和计算都非常复杂。可以通过核函数忽略特征定义和映射函数的计算，从而极大地简化了计算过程。但核函数要求必须是可交换并且是半正定的，体现在具体计算上的基本要求就是输入特征的计算只能以内积形式体现。因此对偶形式保证了可以采用核函数进行计算，从而进一步可以解决非线性分类问题。

例题:编程实现感知机算法的对偶形式求感知机模型。

```
01 def dual(data,alpha,b,lr):                                                    # 数据, alpha, 偏置, 学习率
02     X = data[:,:-1]
03     Y = data[:,-1]
04     count = 0
05     w= [0, 0]
06     ret = [[0,None,alpha.copy(),0]]
07     error_point = not None
08     Gram_matrix = X@X.T                                                       # 构造Gram矩阵，@是点积
09     while error_point:
10         hyperplane(count,X,Y,w,b)
11         error_point = None
12         for i in range(len(Y)):
13             if Y[i]*((alpha*Y*Gram_matrix[i]).sum() + b) <= 0: # 误分类
14                 alpha[i] += lr                                                # 更新alpha
15                 b += Y[i]                                                     # 更新偏置
16                 error_point = f'x{i+1}'                                       # 记录误分类点
17                 break
18         count += 1
19         w = ((alpha*Y)[:,np.newaxis]*X).sum(0)            # 绘图需要，与计算无关
20         if(error_point):
21             ret.append([count,error_point,alpha.copy(),b,w,f'{w[0]}*x1+{w[1]}*x2+{b}'])
22     return ret
```

此处参数*w*的计算仅为绘图需要，在算法的计算过程中不需要计算*w*。在记录*alpha*的时候，因为*alpha*本身是列表，形成了列表嵌套，因此必须使用copy函数。否则最终结果中只会记录一个相同的引用地址，导致结果错误。

最后，读取数据，调用以上函数完成感知机算法的分类。

```
01 data=pd.read_csv('example1.1.csv',header=None,encoding='utf8').values
02 alpha,b,lr = np.zeros(3), 0, 1
03 ret = dual(data,alpha,b,lr)
04 df = pd.DataFrame(ret,columns=['迭代次数','误分类点','α','b','w','模型'])
05 print(df)
```

	代次	次数	误分类点	α	b	w	模型
0	0	None	[0.0, 0.0, 0.0]	0	None		None
1	1	x1	[1.0, 0.0, 0.0]	1	[3.0, 3.0]		3.0*x1+3.0*x2+1
2	2	x3	[1.0, 0.0, 1.0]	0	[2.0, 2.0]		2.0*x1+2.0*x2+0
3	3	x3	[1.0, 0.0, 2.0]	-1	[1.0, 1.0]		1.0*x1+1.0*x2+-1
4	4	x3	[1.0, 0.0, 3.0]	-2	[0.0, 0.0]		0.0*x1+0.0*x2+-2
5	5	x1	[2.0, 0.0, 3.0]	-1	[3.0, 3.0]		3.0*x1+3.0*x2+-1
6	6	x3	[2.0, 0.0, 4.0]	-2	[2.0, 2.0]		2.0*x1+2.0*x2+-2
7	7	x3	[2.0, 0.0, 5.0]	-3	[1.0, 1.0]		1.0*x1+1.0*x2+-3

比较每步得到的模型，感知机的对偶形式和原始形式虽然计算过程不同，但是结果相同。

13.2 K近邻法

K近邻法(K-NN) 是一种基本分类与回归方法。本节只讨论K近邻法的分类问题。

13.2.1 K近邻法

K近邻算法简单、直观，没有显式的学习过程。对于给定的训练数据集，新输入实例在训练数据集中找到与该实例最邻近的k个实例，将这k个实例多数属于的类别定义为新输入实例的类别。它的缺点是计算量比较大，k值很小时容易受异常点影响；k值很大时，容易受数量波动影响。

算法**13.2.1 (K**近邻法**)**

输入：训练数据集$\{(x_1, y_1), (x_2, y_2)\dots, (x_N, y_N)\}$，其中$x_i \in R^n$，$y_i \in \{c_1, c_2, \dots, c_K\}$，$i = 1, 2, \dots, N$；新实例特征向量$x$：

输出：实例x所属的类别y。

(1) 根据给定的距离度量，在训练集中找出与x最邻近的k个点，涵盖这k个点的x的邻域记作$N_k(x)$；

(2) 在$N_k(x)$中根据分类决策规则（如多数表决）决定x的类别y：

$$arg \sum_{x_i \in N_k(x)} I(y_i = c_j), \quad i = 1, 2, \cdots, N; j = 1, 2, \cdots, K$$

其中I为指示函数，当条件成立时取值为1，否则为0：

$$(x) = \{1,\ y = c_j\ 0,\ y \neq c_j$$

■

例题:使用K近邻算法求新输入实例的类别。

```
01 import numpy as np
02 from collections import Counter
03
04 def classify(X, data, labels, k):                                    #输入实例，数据集, 标签，k近邻
05     distances = np.linalg.norm(X-data,2,axis=1)          # L2范数求欧式距离
06     sortedIdx = distances.argsort()                      # 距离排序
07     k_class=[labels[sortedIdx[i]] for i in range(k)]     # 最近的k个类别
08     Class = Counter(k_class).most_common(1)              # 标签最多的类别
09     return Class[0][0]
10 c = classify([0.7,0.8], np.array([[1.0,1.1],[1.0,1.0],[0,0],[0,0.1]]), ['A','A','B','B'], 3)
11 print(c)
```

```
A
```

从以上代码中可以看到，新实例需要跟所有训练集中的实例进行距离计算，当数据量比较大时，是一项非常耗时的操作。计算量庞大是K近邻法的一个重要缺陷。

13.2.2 构造kd树

为了解决K近邻法计算量过大的问题。首先可以将训练数据集构造成一个kd树。利用kd树的特性, 快速定位距离新输入实例最近的k个实例。kd树是一个对k维空间进行划分的二叉树。构造kd树相当于不断地用垂直于坐标轴的超平面将k维空间切分，构成一系列的k维超矩形区域。kd树的每个结点对应于一个k维超矩形区域。

算法**13.2.2 (**构造平衡**kd**树**)**

输入：训练数据集$\{(x_1,\ y_1),\ (x_2,\ y_2)\ldots,(x_N, y_N)\}$, 其中

$x_i = (x_i^{(1)}, x_i^{(2)}, \ldots x_i^{(n)})^T$, $y_i \in \{c_1, c_2, \ldots, c_K\}$, $i \in [1,\ N]$; $x_i^{(j)}$表示第i个样本的第j维特征：

输出: kd树。

(1) 构造根结点，根结点对应于包含训练数据集的k维空间的超矩形区域。

选择$x^{(1)}$为坐标轴，以训练集中所有实例的$x^{(1)}$坐标的中位数为切分点，将根结点对应的超矩形区域切分为两个子区域。切分面为通过切分点并与坐标轴$x^{(1)}$垂直的超平面。

由根结点生成深度为1的左、右子结点；左子结点对应坐标$x^{(1)}$小于切分点的子区域, 右子结点对应于坐标$x^{(1)}$大于切分点的子区域。

将落在切分超平面上的实例点保存在根结点。

(2) 对深度为j的结点，选择$x^{(l)}$为切分的坐标轴，$l= j(mod\ k)\ +\ 1$。以该结点的区域中所有实例的$x^{(l)}$坐标的中位数为切分点，将该结点对应的超矩形区域切分为两个子区域。由该结点生成深度为j+1的左、右子结点：左子结点对应坐标$x^{(l)}$小于切分点的子区域，右子结点对应坐标$x^{(l)}$大于切分点的子区域。

将落在切分超平面上的实例点保存在该结点。

(3) 重复步骤(2)，直到两个子区域没有实例存在时停止。从而形成kd树的区域划分。■

例题：构造输入数据集的平衡kd树。

首先，构造数的节点。

```
01 import numpy as np
02 import pandas as pd
03 from matplotlib import pyplot as plt
04
05 class Node(object):
06     def __init__(self,data,parent=None,left=None,right=None):
07         self.data = data
08         self.left = left
09         self.right = right
10         self.parent = parent
11     def is_leaf(self):
12         return self.left is None and self.right is None
```

然后构造kd树的类。

```
01 class kdTree(object):
02     def __init__(self,data,begin_dim=0):                    # 数据和开始维度
03         self.dataNum = 0
04         self.root = self.buildKdTree(data,begin_dim) # 树保留在根节点root中
05         self.begin_dim = begin_dim                           # 开始构造的维度
06
07     def buildKdTree(self, data, dim=0, parentNode=None):
08         data_len,dim_len = data.shape                        # 训练集的样本数，数据的维数
09         dim_len -= 1                                                    # 最后一维是
样本的类别
10         if data_len==0: return None
11         median = sorted(data[f'c{dim}'])[data_len//2] # 取中位数
12         root_data = data[data[f'c{dim}'] == median]        # 与中位数相等的值保留在当前节点
13         root = Node(root_data,parentNode)
14         if len(root_data)!=len(data):                        # 剩余数据分配给左右两个子节点
15             left_data = data[data[f'c{dim}'] < median]
16             root.left = self.buildKdTree(left_data,(dim+1)%dim_len,root)
17             right_data = data[data[f'c{dim}'] > median]
18             root.right = self.buildKdTree(right_data,(dim+1)%dim_len,root)
19         return root
```

在取中位数时，没有使用Numpy的median函数，是因为当数据量为偶数时，中位数为最近两个数据的平均值，而此处代码保证至少一个实例落在分割面上。

为了清晰的理解数据，可以将数据绘制出来（仅支持二维数据）。

```
01 class kdTree(object):
02     ……
03     def plot2d(self, data):
         # 为2维数据画图
04       if len(data.columns)-1>2: return
05       def draw_node(node,min,max,dim):
06           if node is None: return
07           plt.scatter(node.data['c0'],node.data['c1'])                    # 画点
08           if node.is_leaf(): return
09           median = node.data.iloc[0,dim]
10           if dim == 0:
11               plt.plot((median,median),(min[1],max[1]))                   # 画分割超平面
12               draw_node(node.left,(min[0],min[1]),(median,max[1]),(dim+1)%2)
13               draw_node(node.right,(median,min[1]),(max[0],max[1]),(dim+1)%2)
14           else:
15               plt.plot((min[0],max[0]),(median,median))                   # 画分隔超平面
16               draw_node(node.left,(min[0],min[1]),(max[0],median),(dim+1)%2)
17               draw_node(node.right,(min[0],median),(max[0],max[1]),(dim+1)%2)
18       min,max = np.min(data.min())-1,np.max(data.max())+1                 # 确定数据范围
19       plt.figure(figsize=(5,5))
20       plt.xlim(min,max)
21       plt.ylim(min,max)
22       draw_node(self.root,(min,min),(max,max),1)
23       plt.show()
```

最后，调用以下代码，构造kd树并进行绘图。

```
data = pd.read_csv('./example2.1.csv',header=None,encoding='utf8')
data.columns = [f'c{i}' for i in range(data.shape[1]-1)]+['flag']                # 为所有列命名，方便调用
tree = kdTree(data,1)
tree.plot2d(data)
```

数据文件example2.1.csv的内容如下，它是一个二维数据，最后一列表示实例的类别。

表13.1 example3.3.csv的内容

2.4	5.0	*A*
6.6	1.5	*B*
4.2	8.1	*A*
9.4	3.0	*B*
8.9	5.5	*A*
1.3	1.1	*B*
1.5	8.5	*A*

绘图结果如下，其中y轴方向大于4的实例点的类别为A，其它为B：

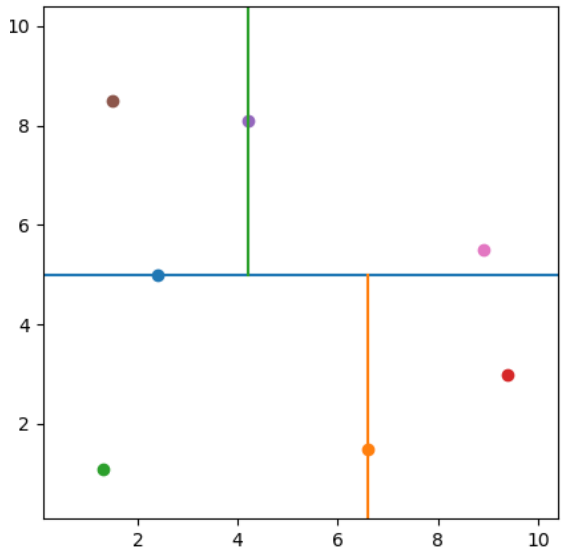


图13.3 example2.1构造kd树的结果

13.2.3 搜索*kd*树

本节介绍如何使用*kd*树进行*k*近邻搜索。*kd*树将训练数据集沿坐标轴方向划分为若干区域。在进行*k*近邻搜索时，首先找到包含新实例点的叶子区域，然后根据最小距离划分超球体，在寻找k近邻的过程中，只判断与超球体相交的区域，因此去除了大量实例点间的距离计算，从而提高了计算效率。超球体指在高维空间下的球体，在每维空间上距离球心点的距离不超过其半径。而*kd*树在每维空间上沿着坐标轴进行划分，因此搜索时超球体与划分区域的相交计算也被简化为一维计算，计算代价非常小。

算法**13.2.3 (**用***kd***树的***k***近邻搜索**)**

输入：已构造的*kd*树，新实例点x；

输出：新实例点x根据多数投票机制形成的类别。

(1) 将*k*近邻列表设置为空，将*kd*树的根节点设置为当前节点*c*。

(2) 从当前节点*c*出发，递归地向下访问*kd*树。若*x*当前维的坐标小于切分点的坐标，则将左子树设置为*c*, 否则将右子树设置为*c*。直到*c*为叶结点为止。

(3) 计算*c*中实例点的类别以及其到*x*距离，与*k*近邻列表中的已有数据合并，挑选其中*k*个最小距离以及其对应类别保留在*k*近邻列表中。将*k*近邻列表中的最大距离设置为以*x*为中心超球体的半径。

(4) 递归地向上回退，将其父结点设置为c, 并进行以下操作:

(a) 如果*k*近邻列表中的数据不足k个，转(c)；

(b) 如果*c*的分隔超平面与超球体不相交，转(4)；

(c) 计算c中实例点的类别以及其到x距离，与k近邻列表中的已有数据合并，挑选其中k个最小距离以及其对应类别保留在k近邻列表中。重置超球体半径。

(d) 对另一子节点重复从(2)开始的步骤。

(5) 当回退到根结点时搜索结束。对k近邻列表中的类别采用多数投票机制得到x的类别。■

通过*kd*树的区域分割，极大的减少了新实例点与训练数据集中已有实例点的距离计算。并且超球体与分割超平面的相交计算被简化为沿着坐标轴的一维比较，简化了计算过程。

例题:对新输入的实例点x通过已构造的的平衡*kd*树判断其类别。

在13.2.2小节构造的*kd*树*kdTree*添加新成员函数*nearest*。

```
01 class kdTree(object):
02     ……
03     def nearest(self, x, k):
04     def find_k(node):
05             nonlocal k_neighbors                                    # 使用父级变量，统一
存储
06             data = node.data.iloc[:,:-1].values
07             flag = node.data.iloc[:,-1].reset_index(drop=True)      # 新数据的类别
08             dis = pd.Series(np.linalg.norm(x-data,2,axis=1),name='dis') # 求欧式距离
09             df = pd.concat([dis,flag],axis=1)                       # 新数据构成
DataFrame
10             k_neighbors = k_neighbors.append(df).nsmallest(k,'dis')
11
12     def visit(node,dim):
13             if node is None: return
14             if node.is_leaf():
15                     return find_k(node)      # 将匹配的叶子节点的数据进行k近邻判定
16             median = node.data.iloc[0,dim]
17             cur,other = (node.left,node.right) if x[dim]<median else (node.right,node.left)
18             visit(cur,(dim+1)%len(x))          # 寻找叶子节点
19             # 如果尚未找到k个近邻，或分隔面与最小距离超球体相交
20             if len(k_neighbors)<k or abs(x[dim]-median)<k_neighbors['dis'].values[-1]:
21             find_k(node)               # 分隔面上的数据是否是k近邻
22             visit(other, (dim+1)%len(x))      # 判别另一个子节点
23
24     k_neighbors = pd.DataFrame(columns=['dis','flag']) # 统一存储近邻列表
25     visit(self.root,self.begin_dim)
26     return k_neighbors['flag'].value_counts().index[0]
27 print(tree.nearest([8.9,4.5],3))                                   # 对新实例点[8.9,4.5]
判断3近邻时的类别
```

```
B
```

其中辅助函数*find_k*判断节点*node*中实例点是否有实例点距离x更近，如果有则置于k近邻列表*k_neighbors*中。第5行中采用关键字*nonlocal*，保障*k_neighbors*在第10行进行更新时，不会被局部变量覆盖。这也是*k_neighbors*与同为父级变量x，k间最大的区别，因为x，k在局部作用域使用的过程中，没有赋值操作，因此不需要显示指定为*nonlocal*。第6-9行将当前节点中实例点的到x的距离和对应的类别构建了一个新的*DataFrame*。第10行将k近邻列表中的已有数据与新数据进行拼接，排序后选择距离最小的k个，形成新的k近邻列表。第7行中的*reset_index*保证第9行拼接时索引的一致性，第8行的参数*name*保证第10行*append*时列标题相同。

函数*visit*的计算逻辑与算法13.2.3完全匹配。第20行中一定要判断k近邻列表中的数据是否已经有*k*个，否则根据当前超球体半径直接判断可能会跳过一些符合条件的实例点。*k*近邻列表保存*k*个最近距离以及对应的类别，按照距离从近到远排序，因此该列表中最后一个距离被认定为超球体半径。*Counter*类的*most_common*函数保存一个类别和频次的列表，按频次从大到小排序。因此第0个元素的第一个分量为*x*的类别。

通过以上程序，采用*kd*树进行k近邻搜索时，不需要计算*x*和所有实例点的距离。当*k*为3时，访问5个实例点；当*k*为2时，访问5个实例点；*k*为1时，仅访问3个实例点。当数据量比较大时，能充分发挥*kd*树的性能优势。而传统的k近邻算法13.2.3每次都要和所有实例点进行距离计算。当最近类别的投票结果相等时，较近的类别会被标记为*x*的类别。第26行采用*most_common*函数进行处理时，因为*k_neighbors*是经过距离从小到大排序的，因此即使两个类别的数量相等，最近的类别会成为*x*的最终类别。例如*k*为2时，最近的两个类别分别为A和B，但类别为A的实例点距离*x*更小，因此最终结果为A。

表13.2 k近邻算法中不同*k*的结果

k近邻	访问实例点数	最终类别
1	3	*A*
2	5	*A*
3	5	*B*

13.3 朴素贝叶斯

朴素贝叶斯法是基于贝叶斯定理的后验概率最大化分类方法。所谓朴素，是指输入数据每个特征维度相对独立。而贝叶斯方法的核心是采用后验概率。该方法实现简单，但训练与预测的效率都很高，属于常见方法。

13.3.1 朴素贝叶斯

朴素贝叶斯的核心思想是逆概率计算。用一个例子讲解其中的主要概念。假定有两个盒子，第一个盒子c_1放了两个白球(w)和两个黑球(b)，第二个盒子c_2放了一个白球(w)和两个黑球(b)。根据极大似然估计，很容易计算$P\left(Y = c_1\right)$=4/7和$P\left(Y = c_2\right) = 3/7$，这就是先验概率。在第一个盒子中取得白球的概率为$P\left(X = w|Y = c_1\right)$=2/4，而在第二个盒子中取得白球的概率为$P\left(X = w|Y = c_2\right)$=1/3，这种在约束条件下的概率称为条件概率。先验概率和条件概率都可以根据已知训练数据集利用极大似然估计进行计算，它们是贝叶斯方法的基础。

联合概率：联合概率是指在多元的概率分布中，个随机变量同时各自条件的概率，例如$P(X^{(1)} = x^{(1)}, X^{(2)} = x^{(2)})$或$P(X^{(1)} = x^{(1)}, Y = c_k)$，$X^{(j)}$表示样本的第$j$维特征，$c_k$表示$Y$第$k$个取值。一个样本可能有多个特征，例如颜色特征，白、黑等；形状特征，球、方块等。联合概率就是指多个特征同时符合条件的概率。

朴素：朴素贝叶斯中的“朴素”，是指样本中每个维度相对独立。假设$X^{(j)}$可能取值有S_j个，$j \in [1, n]$，Y可取值有K个，那么参数个数为$K\prod_{j=1}^{n} S_j$，这在实际计算中非常复杂。因此朴素贝叶斯建立在条件概率分布的条件独立性假设基础之上，即：

$$P(X = x) = P\left(X^{(1)} = x^{(1)}, X^{(2)} = x^{(2)}, ..., X^{(n)} = x^{(n)}\right) = \prod_{j=1}^{n} P(X^{(j)} = x^{(j)})$$

$$P(X|Y) = P\left(X^{(1)} = x^{(1)}, X^{(2)} = x^{(2)}, ..., X^{(n)} = x^{(n)}|Y = c_k\right) = \prod_{j=1}^{n} P(X^{(j)} = x^{(j)}|Y = c_k)$$

这一假设使朴素贝叶斯变得简单，但有时会牺牲一定的分类准确率。

条件概率公式:条件概率$P(X|Y)$表示Y成立时，X成立的概率, 其公式为：

$$P(X = x|Y = c_k) = \frac{P(X=x,Y=c_k)}{P(Y=c_k)}$$

简写为：

$$P(X|Y) = \frac{P(X,Y)}{P(Y)}$$

即条件概率等于联合概率除以先验概率。将其变形可得：

$$P(X = x, Y = c_k) = P(Y = c_k)P(X = x|Y = c_k) = P(Y = c_k)\prod_{j=1}^{n} P(Y = c_k)$$

简写为：

$$P(X, Y) = P(Y)P(X|Y) = P(X)P(Y|X)$$

全概率公式:假设Y可取值有K个, c_k表示Y第k个可能取值，则：

$$P(X = x) = \sum_{k=1}^{K} P(X = x, Y = c_k)$$
$$= \sum_{k=1}^{K} P(Y = c_k)P(X = x|Y = c_k)$$

与联合概率相对应，$P(X = x)$被称为边缘概率。

导致$X = x$发生可能有各种原因$\{c_1, c_2, ..., c_K\}$, 但$X = x$发生的概率$P(X = x)$是全部原因引起$X = x$发生的概率$P(X = x, Y = c_k)$的总和。全概率公式的意义在于它将一个复杂事件转换为简单事件的总和。

贝叶斯公式:由以上条件概率公式和全概率公式, 可得贝叶斯公式：

$$P(X = x) = \frac{P(X=x,Y=c_k)}{P(X=x)} = \frac{P(Y=c_k)P(X=x|Y=c_k)}{\sum_{k=1}^{K} P(Y=c_k)P(X=x|Y=c_k)}$$

其中分子部分由条件概率公式推导，分母部分由全概率公式推导。

逆概率计算：在前面的例子中, 已知选择的盒子(Y), 然后求某种颜色特征(X)出现的概率，即求$P(X)$, 称为先验概率。贝叶斯方法是逆概率计算，由已经发生的事情，推断其发生的原因，即求$P(Y|X)$, 称为后验概率。在上面例子中，如果知道取出一个白球(X), 求其属于某个盒子(Y)的概率。从这个例子中可以看出，先验概率和条件概率可以较容易采用极大似然估计进行计算，但

后验概率计算比较困难。贝叶斯公式的伟大之处在于，它将后验概率的求解，转换为先验概率和条件概率的计算，从而降低了计算的难度。

通过以上概念的铺垫，贝叶斯分类器可表示为：

$$y = f(x) = \arg arg\, P(X = x) = \arg arg\, \frac{P(Y=c_k)P(X=x|Y=c_k)}{P(X=x)}$$

因为分母与c_k无关, 因此:

$$y = f(x) = \arg arg\, P(Y = c_k)P(Y = c_k)$$

$$= \arg arg\, P(Y = c_k)\prod_{j=1}^{n} P(Y = c_k)$$

13.3.2 朴素贝叶斯的极大似然估计

算法**13.3.1 (**朴素贝叶斯算法**)**

输入：训练数据集$\{(x_1, y_1), (x_2, y_2)\dots, (x_N, y_N)\}$, 其中

$x_i = (x_i^{(1)}, x_i^{(2)}, \dots x_i^{(n)})^T$, $y_i \in \{c_1, c_2, \dots, c_K\}$, $i\in[1, N]$; $x_i^{(j)}$表示第i个样本的第j维特征，

$x_i^{(j)}\in\{a_{j1}, a_{j2}, \dots a_{jS_j}\}$, a_{jt}是第j个特征的第t个可能取值，$j\in[1, n]$, $t\in[1, S_j]$, S_j表示j个特征的可能取值个数；实例x;

输出：实例x的类别y。

(1) 计算先验概率及条件概率

$$P(Y = c_k) = \frac{\sum_{i=1}^{N} I(y_i=c_k)}{N}$$

$$P(X^{(j)} = a_{jl}|Y = c_k) = \frac{\sum_{i=1}^{N} I(x_i^{(j)}=a_{jl}, y_i=c_k)}{\sum_{i=1}^{N} I(y_i=c_k)}$$

(2) 对于给定的实例x计算

$$P(Y = c_k)\prod_{j=1}^{n} P(X^{(j)} = x^{(j)}|Y = c_k)$$

(3) 确定x的类别y:

$$argmax_{c_k} P(Y = c_k)\prod_{j=1}^{n} P(X^{(j)} = x^{(j)}|Y = c_k) \blacksquare$$

例题:使用朴素贝叶斯算法求新输入实例的类别。

训练数据如下：

表13.3 训练数据

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$X^{(1)}$	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3
$X^{(2)}$	S	M	M	S	S	S	M	M	L	L	L	M	M	L	L
Y	-1	-1	1	1	-1	-1	-1	1	1	1	1	1	1	1	-1

```
01 import pandas as pd
02 import numpy as np
03
04 class Bayes:
05     def __init__(self,data,lamda=0):
06         self.data = data
07         self.lamda = lamda
08     def p(self,y_val,feature='y',x_val=None):
09         S = len(self.data[feature].unique())                                    #贝叶
斯估计λ的倍数
10         if feature=='y': val,data = y_val,self.data                             #为求概率做准备
11         else: val,data = x_val,self.data[self.data['y']==y_val]                 #为求条件概率做准备
12         return (data[feature].value_counts()[val]+self.lamda)/(len(data)+S*self.lamda)
13     def list_x_pro(self):
                   #列出所有条件概率
14         for y in self.data['y'].unique():                                       #y的
所有可能取值
15             for x_name in self.data.columns[:-1]:
16                 for x in self.data[x_name].unique():                    #对应特征所有可能取值
17                     print(f'P({x_name}={x}|y={y})={self.p(y,x_name,x):.4f}',end='\t')
18                 print()
19     def list_y_pro(self):
                   #列出所有先验概率
20         for y in self.data['y'].unique():
21             print(f'P(y={y})={self.p(y):.4f}')                                  #计算先验概
率
22     def __call__(self, x):
         #朴素贝叶斯方法分类
23         Y = self.data['y'].unique()
24         ret = {}
25         for y in Y:
26             ret[y]= self.p(y)                                                   #先验
概率
27             for i,x_name in enumerate(self.data.columns[:-1]):
28                 ret[y] *= self.p(y,x_name,x[i])                             #连乘条件概率
29             print(f'P(y={y})P(X={x}|y={y}))={ret[y]:.4f}')
30         return Y[np.argmax(ret.values())]                                       #后验
概率最大的类别
31
32 data = pd.read_csv('./example3.1.csv',encoding='utf8')
33 b = Bayes(data,0)
34 b.list_y_pro()
```

```
35 b.list_x_pro()
36 x = [2,'S']
37 print(f"The class of {x} is ",b(x))
```

```
P(y=-1)=0.4000
P(y=1)=0.6000
P(x1=1|y=-1)=0.5000          P(x1=2|y=-1)=0.3333     P(x1=3|y=-1)=0.1667
P(x2=S|y=-1)=0.5000          P(x2=M|y=-1)=0.3333     P(x2=L|y=-1)=0.1667
P(x1=1|y=1)=0.2222           P(x1=2|y=1)=0.3333      P(x1=3|y=1)=0.4444
P(x2=S|y=1)=0.1111           P(x2=M|y=1)=0.4444      P(x2=L|y=1)=0.4444
P(y=-1)P(X=[2, 'S']|y=-1))=0.0667
P(y=1)P(X=[2, 'S']|y=1))=0.0222
The class of [2, 'S'] is  -1
```

以上代码中，最重要的是函数*p*，它在*x_val*为*None*时，计算*Y*的先验概率，否则计算条件概率。重载函数*__call__*依靠函数*p*计算的先验概率和条件概率求解贝叶斯分类器的结果。

在以上代码中，类成员变量*lamda*默认为0，表示极大似然估计。但在求解过程中，如果要估计的概率值为0，就会影响后验概率的计算结果，导致分类产生偏差。因此在公式中加入一个正数λ，用于解决这个问题，这样极大似然估计就转换为贝叶斯估计。尤其当λ=1时，称为拉普拉斯平滑。相应的先验概率和条件概率公式修改如下：

$$P_\lambda(Y=c_k)=\frac{\sum_{i=1}^{N} I(y_i=c_k)+\lambda}{N+K\lambda}$$

$$P_\lambda\left(Y=c_k\right)=\frac{\sum_{i=1}^{N} I\left(x_i^{(j)}=a_{jt},y_i=c_k\right)+\lambda}{\sum_{i=1}^{N} I(y_i=c_k)+S_j\lambda}$$

其中*I*为指示函数。

注意，贝叶斯方法和贝叶斯估计是完全不同的两个概念，应用场景也完全不同。

从公式中可以很容易得到$\sum_{i=1}^{K} P_\lambda(Y=c_k)=1$，且$\sum_{i=1}^{S_j} P_\lambda\left(Y=c_k\right)=1$，表明贝叶斯估计确实是一种概率分布。将上面代码中成员变量*lamda*修改为1后，运行结果如下：

```
01 data = pd.read_csv('./ example3.1.csv',encoding='utf8')
02 b = Bayes(data,1)
03 b.list_y_pro()
04 b.list_x_pro()
05 x = [2,'S']
06 print(f"The class of {x} is ",b(x))
```

```
P(y=-1)=0.4118
P(y=1)=0.5882
P(x1=1|y=-1)=0.4444 P(x1=2|y=-1)=0.3333 P(x1=3|y=-1)=0.2222
P(x2=S|y=-1)=0.4444 P(x2=M|y=-1)=0.3333 P(x2=L|y=-1)=0.2222
P(x1=1|y=1)=0.2500  P(x1=2|y=1)=0.3333  P(x1=3|y=1)=0.4167
P(x2=S|y=1)=0.1667  P(x2=M|y=1)=0.4167  P(x2=L|y=1)=0.4167
P(y=-1)P(X=[2, 'S']|y=-1))=0.0610
P(y=1)P(X=[2, 'S']|y=1))=0.0327
```

```
The class of [2, 'S'] is  -1
```

13.4 决策树

决策树是一种基本的分类与回归方法。决策树是一种树形结构，其中每个内部节点表示一个属性上的测试，每个分支代表一个测试输出，每个叶节点代表一种类别。决策树学习通常包括3个步骤：特征选择、决策树的生成和决策树的修剪。从根节点出发，在每一级节点处，用选定的特征划分为多个子树，直到叶子节点为止。为了优化决策树，需要对决策树进行剪枝。决策树算法的核心是选择哪个特征进行划分，如何进行划分。本节主要介绍ID3、C4.5和CART三种算法，分别采用信息增益、信息增益比和基尼指数进行划分。

决策树的生成基于基于有限个值的训练数据集，以下给出一个示例数据。

表13.4.1 示例数据example4.1

age	work	house	load	y
young	no	no	common	no
young	no	no	good	no
young	yes	no	good	yes
young	yes	yes	common	yes
young	no	no	common	no
mid	no	no	common	no
mid	no	no	good	no
mid	yes	yes	good	yes
mid	no	yes	best	yes
mid	no	yes	best	yes
old	no	yes	best	yes
old	no	yes	good	yes
old	yes	no	good	yes
old	yes	no	best	yes
old	no	no	common	no

对于同一个数据集，采用不同的特征进行划分，特征选择不同的顺序，以及不同的划分准则，都可以生成不同的决策树。

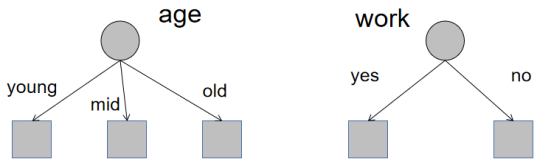


图13.4 不同特征决定不同的决策树

13.4.1 信息增益构建决策树

信息增益的核心概念是熵和条件熵。设X和Y表示个数为N的离散随机变量。

熵：将X等于x_i的概率表示为$P(X = x_i) = p_i$，则随机变量X的熵表示为

$$H(X) = -\sum_{i=1}^{N} p_i log_2 p_i$$

若p_i=0, 则定义0ln0=0。$H(X)$也可以记为$H(p)$。熵越大，表示随机变量的不确定性越大。从熵的定义可知，0≤$H(p) \le n$。当随机变量只有0,1两个值时，设$P(X = 1) = p$, 则熵为：

$$H(X) = -plog_2(p) - (1 - p)log_2(1 - p)$$

条件熵:设有随机变量(X,Y)，联合分布概率为$P\left(X = x_i, Y = y_j\right) = p_{ij}$, 随机变量X给定条件下随机变量Y的条件熵定义为：

$$H(Y|X) = -\sum_{i=1}^{N} p_i H(Y|X = x_i)$$

信息增益:表示已知特征X的信息而使得类Y的信息的不确定性减少的程度。在决策树中, 信息增益等价于训练数据集中类与特征的互信息$g(Y, X)$。

$$g(Y, X) = H(Y) - H(Y|X)$$

设数据集有N个样本，其中Y有K个可能取值$\{c_1, c_2, ..., c_K\}$, 每个样本X包含n个特征, 其中第j维特征$X^{(j)}$可能取值有S_j个, j∈$[1, n]$。用$|c_k|$表示属于类c_k的样本个数，$|X_t^{(j)}|$表示特征$X^{(j)}$等于第t个值的样本个数，其中t∈$[1, S_j]$;$|X_{tk}^{(j)}|$表示特征$X^{(j)}$等于第t个值，且属于类c_k的样本个数。则数据集的熵为：

$$H(Y) = -\sum_{k=1}^{K} \frac{|c_k|}{N}\frac{|c_k|}{N}$$

$$H\left(X^{(j)}\right) = -\sum_{t=1}^{S_j} \frac{|X_t^{(j)}|}{N}\frac{|X_t^{(j)}|}{N}$$

特征$X^{(j)}$对数据集的条件熵为：

$$H\left(Y|X^{(j)}\right) = -\sum_{t=1}^{S_j} \frac{|X_t^{(j)}|}{N}\sum_{k=1}^{K} \frac{|X_{tk}^{(j)}|}{N|X_t^{(j)}|}\frac{|X_{tk}^{(j)}|}{|X_t^{(j)}|}$$

特征$X^{(j)}$对数据集的信息增益为：

$$g\left(Y, X^{(j)}\right) = H(Y) - H(Y|X^{(j)})$$

特征$X^{(j)}$对数据集的信息增益比为：

$$g_R\left(Y, X^{(j)}\right) = \frac{g\left(Y, X^{(j)}\right)}{H\left(X^{(j)}\right)}$$

算法13.4.1 (ID3算法(信息增益)或C4.5算法(信息增益比))

输入：训练数据集$\{(x_1, y_1), (x_2, y_2)\dots, (x_N, y_N)\}$, 其中

$x_i = (x_i^{(1)}, x_i^{(2)}, \dots x_i^{(n)})^T$, $y_i \in \{c_1, c_2, \dots, c_K\}$, $i \in [1, N]$; $X^{(j)} \in \{a_{j1}, a_{j2}, \dots a_{jS_j}\}$, a_{jt}是第j个特征的第t个可能取值阈值ε;

输出：决策树T

(1) 若数据集中Y只有一个类别c_k, 则T为单结点树，并将c_k作为该结点的类标记，返回T;

(2) 如果数据集中没有可以区分的特征, 则T为单结点树，将Y中数量最多的类c_k作为该结点的类标记，构建结束；

(3) 计算数据集中X各特征对产生的信息增益（比），选择信息增益（比）最大的特征$X^{(j)}$;

(4) 如果$X^{(j)}$的信息增益(比)小于阈值ε, 则置T为单结点树，并将Y中数量最多的类c_k作为该结点的类标记，构建结束；

(5) 否则，对$X^{(j)}$的每一可能值a_{jt}, 依$X^{(j)} = a_{jt}$将数据集分割为若干非空子集，将每个子集去除$X^{(j)}$特征, 递归地调用(1)~(5)，得到树T, 构建结束。■

```
01 import pandas as pd
02 from math import log2
03
04 class DecisionTree:
05     def __init__(self,train,algorithm='ID3',epsilon = 0.1):
06         self.columns = train.columns[:-1]
           #不包括y
07         self.build(train,algorithm,epsilon)
                   #算法和阈值
08     def build(self,train,algorithm,epsilon = 0.1):
09         def split(data):
10             def entropy(data,col='y'):
           #熵
11                 func = lambda sub:len(sub)/len(data)*log2(len(sub)/len(data))
12                 return -data.groupby(col).apply(func).sum()
13             def c_entropy(data,col):
           #条件熵
14                 func = lambda sub:len(sub)/len(data)*entropy(sub)
15                 return data.groupby(col).apply(func).sum()
16             def ID3(col):
                       #信息增益
17                 return entropy(data)-c_entropy(data,col)
18             def C45(col):
                       #信息增益比
19                 if len(data)==0: return 0
20                 return ID3(col)/entropy(data,col)
21
22             if len(data['y'].unique())==1: return data['y'].iloc[0]                    #算法13.4.1(1)
23             if len(data.columns)==1: return data['y'].value_counts().index[0]#算法13.4.1(2)
```

```
24          delta = data.drop('y',axis=1).apply(eval(algorithm))                    #每列信息变
化
25          col = delta.idxmax()
        #算法13.4.1(3)
26          if delta[col]<epsilon:
        #算法13.4.1(4)
27              return data['y'].value_counts().index[0]
28          ret = {v:split(s.drop(col,axis=1)) for v,s in data.groupby(col)}     #算法13.4.1(5)
29          return {'col':col,**ret}            #形成节点，记录分类的列，每个值形成一个子节点
30      self.tree = split(train)
31    def fit(self,lst):
32      def predict(x,node):
33          if isinstance(node,dict):
                #不是叶节点
34              return predict(x,node[x[node['col']]])                              #选择子节点
35          return node
                #叶节点
36      df = pd.DataFrame(lst,columns = self.columns)
37      return df.apply(predict,axis=1,args=(self.tree,)).values                        #对每
个值预测
38
39 data = pd.read_csv('./example4.1.csv',encoding='utf8')
40 model = DecisionTree(data,'ID3')
41 print('ID3',model.tree)
42 print 'predicted', (model.fit([['mid','yes','yes','best'],['old','no','no','common']]))
43 model = DecisionTree(data,'C45')
44 print('C4.5',model.tree)
45 print('predicted', model.fit([['mid','yes','yes','best'],['old','no','no','common']]))
```

```
ID3 {'col': 'house', 'no': {'col': 'work', 'no': 'no', 'yes': 'yes'}, 'yes': 'yes'}
predicted ['yes' 'no']
C4.5 {'col': 'house', 'no': {'col': 'work', 'no': 'no', 'yes': 'yes'}, 'yes': 'yes'}
predicted ['yes' 'no']
```

13.4.2 单特征CART回归树

分类与回归树（CART）模型是在给定输入随机变量条件下输出随机变量*Y*的条件概率分布的学习方法。CART假设决策树是二叉树，内部结点特征的取值为“是”和“否”。这样的决策树等价于递归地二分每个特征，将输入空间即特征空间划分为有限个单元，并在这些单元上确定预测的概率分布。

本小结主要讨论*X*为单特征时构建回归树的情况，划分方法相对比较简单，但体现CART模型的基本思想。

表13.4.2 示例数据example4.2

x	y
1	4.5
2	4.75
3	4.91
4	5.34

5	5.8
6	7.05
7	7.9
8	8.23
9	8.7
10	9

算法**13.4.2** 单特征**CART**回归树

输入：训练数据集$\{(x_1, y_1), (x_2, y_2)\dots, (x_N, y_N)\}$, 其中$x_i \in R$, $y_i \in R$, $i \in [1, N]$;

输出：CART回归树T

在训练数据集所在的输入空间中，递归地将每个区域划分为两个子区域并决定每个子区域上的输出值，构建二叉决策树:；

(1) 对数据集选择最优切分点s, 将数据集划分为两个子区域$R_1(s)$和$R_2(s)$:遍历所有切分点，选择使下式达到最小值的切分点s:

$$min_s\left[min_{c_1}\sum_{x_i \in R_1(s)}(y_i - c_1)^2 + min_{c_2}\sum_{x_i \in R_1(s)}(y_i - c_2)^2\right]$$

$$R_1(s) = \{x \mid x, s\}, \quad R_2(s) = \{x \mid x > s\}$$

容易证明，每个子区域的最优切分点c_m, $m = 1, 2$就是该区域的平均值。

(2) 继续对两个子区域调用步骤(1)，直至满足停止条件。

(3) 将输入空间划分为M个区域R_1, R_2,… , R_M, 生成决策树:

$$f(x) = \sum_{m=1}^{M} \hat{c}_m I(x \in R_m)$$

```
01 import pandas as pd
02
03 class RegressTree:
04     def __init__(self,train) -> None:
05         self.bins = list(train['x'].agg([min,max]).values)
06         self.build(train.sort_values('x'))                    #求解分割点要求数据必须有
序
07     def build(self,train):
08         def split(data):
09             var =lambda R:((R['y']-R['y'].mean())**2).sum()                #方差
10             se = lambda p:data.groupby(data['x']>p).apply(var).sum()      #切分后方差和
11             s = data['point'][data['point'].apply(se).idxmin()]                #最优切分点
12             self.bins = sorted(self.bins+[s])                                  #记录
所有切分点
13         train['point'] = (train['x']-train['x'].diff()/2)
```

```
14        for _ in range(2):
                #分割两个层次
15            train.groupby(pd.cut(train['x'],self.bins)).apply(split)
16        self.cm = train.groupby(pd.cut(train['x'],self.bins)).mean()['y']        #各区域平均值c_m
17    def fit(self,lst):
18        return self.cm[pd.cut(lst,self.bins)].values                              #根据所属区域计算
19
20 data = pd.read_csv('./example4.2.csv',encoding='utf8')
21 model = RegressTree(data)
22 print('splitted points',model.bins)
23 print('predicted',model.fit([7.2,7.8,8.8]))
```

```
splitted points [1, 3.5, 5.5, 7.5, 10]
predicted [7.475      8.64333333      8.64333333
```

13.4.3 多特征CART回归树

本小结主要讨论X为多特征时构建回归树的情况。要求所有值都为数值类型，如果是字符串等非数值类型，可以使用One-Hot等方法转换为数值类型。以下为示例数据，其中前三列程序员(coder)、设计师(desiner)和教师(teacher)采用One-Hot编码，对应位为1，其他为0。

表13.4.3 示例数据example4.3

coder	desiner	teacher	age	y
1	0	0	22	20
1	0	0	23	26
1	0	0	29	30
0	1	0	23	12
0	1	0	25	14
0	0	1	32	12
0	0	1	33	16
0	0	1	39	20

算法**13.4.3** 多特征**CART**回归树

输入：训练数据集$\{(x_1, y_1), (x_2, y_2)\dots, (x_N, y_N)\}$, 其中$x_i \in R^n$, $y_i \in R$, $i \in [1, N]$;

输出：CART回归树T

(1) 若数据集中Y只有唯一值，停止划分，返回Y的平均值。

(2) 对所有特征做如下计算：

(a) 如果该特征只有唯一值，不对该特征进行划分；

(b) 循环遍历该特征所有候选切分点，按以下公式寻找最小切分点，以及对应的最小值。

$$min_s\left[min_{c_1}\sum_{x_i \in R_1(s)}(y_i - c_1)^2 + min_{c_2}\sum_{x_i \in R_1(s)}(y_i - c_2)^2\right]$$
$$R_1(s) = \{x|x \leqslant s\}, \quad R_2(s) = \{x|x > s\}$$

(3) 对数据集选择最优切分特征j和最优切分点s，将数据集划分为两个子区域$R_1(j,s)$和$R_2(j,s)$，每个子区域的最优切分点c_m，$m=1,2$就是该区域的平均值。

(4) 如果所有特征都不能进行划分，则返回Y的平均值。

(5) 如果根据最优切分点划分后，信息增益不满足阈值，则返回Y的平均值。

(6) 反复执行以上步骤，最终将输入空间划分为M个区域R_1，R_2，…，R_M，生成决策树：

$$f(x)=\sum_{m=1}^{M}\hat{c}_m I\left(x\in R_m\right)\blacksquare$$

```
01 import pandas as pd
02 import numpy as np
03
04 class RegressTree:
05     def __init__(self,train,delta=10):                                    #delta方差减小程度
06         self.columns = train.columns[:-1]
07         self.build(train,delta)
08     def build(self,train,delta):
09         def split(data):
10             def each_col(col):
11                 X = pd.Series(sorted(data[col.name].unique()))
12                 if len(X)==1: return np.inf,X[0]                #特征只有唯一值
13                 s = (X-X.diff()/2)[1:]                                    #候选分割点
14                 se = lambda p,col:data.groupby(data[col]>p).apply(var).sum()#左右方差和
15                 series = s.apply(se,args=(col.name,))           #特征所有切分点的方差和
16                 return series.min(),s[series.idxmin()]          #最小值和切分点
17             if len(data['y'].unique())==1:  return data['y'].mean() #所有y都相同
18             var =lambda R:((R['y']-R['y'].mean())**2).sum()          #方差和
19             ret = data.drop('y',axis=1).apply(each_col)              #所有特征最小值和切分点
20             if abs(var(data)-ret.iloc[0,:].min())<delta: return data['y'].mean() #增益不足
21             col = ret.iloc[0,:].idxmin()                            #最优特征
22             val = ret[col][1]                                       #最优特征的最优切分点
23             if val==np.inf:   return data['y'].mean()           #所有特征都不可分
24             op = ['<=','>']
25             node = {f'{col}{op[i]}{val}':split(sub) for i,sub in data.groupby(data[col]>val)}
26             return {'col':col,'val':val,**node}                     #分割值、分割列、子节点
27         self.tree = split(train)
28         print(self.tree)
29     def fit(self,lst):
30         def predict(x,node):
31             if isinstance(node,dict):                               #不是叶节点
32                 op,col,val = ['<=','>'],node['col'],node['val']
33                 return predict(x,node[f'{col}{op[x[col]>val]}{val}'])   #选择子节点
34             return node                                             #叶节点
```

```
        df = pd.DataFrame(lst,columns = self.columns)
        return df.apply(predict,axis=1,args=(self.tree,)).values        #预测每个实例

data = pd.read_csv('./example4.3.csv',encoding='utf8')
model = RegressTree(data)
print('predicted',model.fit([[0,0,1,35],[0,1,0,24],[1,0,0,24]]))
```

```
{'col': 'coder', 'val': 0.5, 'coder<=0.5': {'col': 'age', 'val': 32.5, 'age<=32.5': 12.666666666666666, 'age>32.5': 18.0}, 'coder>0.5': {'col': 'age', 'val': 22.5, 'age<=22.5': 20.0, 'age>22.5': 28.0}}
predicted [18.         12.66666667         28.        ]
```

13.4.4 CART分类树

CART分类树用基尼指数选择最优特征，同时决定该特征的最优二值切分点。

基尼指数:分类问题中，假设有K个类，样本点属于第k类的概率为p_k,则概率分布的基尼指数定义为

$$(p) = \sum_{k=1}^{K} p_k\left(1 - p_k\right) = 1 - \sum_{k=1}^{K} p_k^2$$

算法**13.4.4 CART**分类树

输入：训练数据集$D = \left\{\left(x_1, y_1\right), \left(x_2, y_2\right)\ldots, \left(x_N, y_N\right)\right\}$;停止条件阈值和叶节点最小样本数；

输出：CART分类树tree

(1) 若数据集中样本数小于指定的最小样本数，停止划分，按照多数投票机制返回类c_k作为该结点的类标记。

(2) 按如下公式计算数据集D的基尼指数, 如果基尼指数小于阈值，停止划分，按照多数投票机制返回类c_k作为该结点的类标记。

$$(D) = 1 - \sum_{k=1}^{K} \left(\frac{\left|C_k\right|}{|D|}\right)^2$$

(3) 对所有特征做如下计算：

(a) 对于任意一个特征A，根据特征A是否取某一可能值a, 将数据集D分割成D_1和D_2两部分；在特征A的条件下, 集合D的基尼指数定义为

$$D_1 = \left\{x^{(A)} = a\right\},\ D_2 = D - D_1$$
$$(D, A) = \frac{\left|D_1\right|}{|D|}\left(D_1\right) + \frac{\left|D_2\right|}{|D|}\left(D_2\right)$$

其中$x^{(A)}$表示x的特征A。

(b) 计算所有特征的所有取值的基尼指数*Gini(D, A)*, 从中选取基尼指数最小的特征A*和取值a*。将数据集划分为两个部分。

(4) 充分执行以上步骤，得到CART分类树。

基尼指数*Gini(D)*表示集合*D*的不确定性，基尼指数*Gini(D, A)*表示经*A=a*分割后集合*D*的不确定性。基尼指数值越大，样本集合的不确定性也就越大，这一点与熵相似。

```
01 import pandas as pd
02
03 class CartTree:
04     def __init__(self,train,min_samples = 2,epsilon = 0.1):
05         self.columns = train.columns[:-1]
06         self.op = ['!=','==']
07         self.build(train,min_samples,epsilon)                                    #指定最小样
本数和阈值
08     def build(self,train,min_samples = 2,epsilon = 0.1):
09         def split(D):
10             def each_a(a,A):                                                     #A所
有值的Gini指数
11                 func = lambda d:len(d)*gini(d['y'])/len(D)              #Gini指数乘以系数
12                 g = (D.groupby(D[A]==a).apply(func)).sum()              #A的Gini指数
13                 return pd.Series([A,a,g],index=['A','a','gini'])
14             def each_A(A):
15                 ret = pd.Series(D[A].unique()).apply(each_a,args=(A,))
16                 return ret.nsmallest(1,'gini').iloc[0,:]                #最优值a和Gini指数
17             gini = lambda y:1-((y.groupby(y).count()/len(y))**2).sum()#Gini指数
18             if len(D)<=min_samples:return D['y'].value_counts().index[0]
19             if gini(D['y'])<epsilon:return D['y'].value_counts().index[0]
20             ret = pd.Series(D.columns.drop('y')).apply(each_A)       #所有特征Gini指数
21             A,a,_ = ret.nsmallest(1,'gini').iloc[0,:]                        #最优特征和最优值
22             node = {f'{A}{self.op[i]}{a}':split(d) for i,d in D.groupby(D[A]==a)}
23             return {'A':A,'a':a,**node}                                      #最优特征、值
和子节点
24         self.tree = split(train)
25     def fit(self,lst):
26         def predict(x,node):
27             if isinstance(node,dict):                                        #不是
叶节点
28                 A,a = node['A'],node['a']
29                 return predict(x,node[f"{A}{self.op[x[A]==a]}{a}"])#选择子节点
30             return node                                                      #叶节
点
31         df = pd.DataFrame(lst,columns = self.columns)
32         return df.apply(predict,axis=1,args=(self.tree,)).values
33 D = pd.read_csv('./ example4.1.csv',encoding='utf8')
34 model = CartTree(D)
35 print(model.tree)
36 print('predicted',model.fit([['mid','yes','yes','best'],['old','no','no','common']]))
```

```
  {'col': 'house', 'val': 'no', 'house!=no': 'yes', 'house==no': {'col': 'work', 'val': 'no', 'work!=no': 'yes',
'work==no': 'no'}}
  predicted ['yes' 'no']
```

13.5 逻辑斯蒂回归

13.5.1 二项逻辑斯蒂回归模型

逻辑斯蒂回归虽然名为“回归”，其实是统计学习中经典的分类方法。二项逻辑斯蒂回归模型由条件概率分布*P(Y|X)*表示，形式为参数化的逻辑斯谛分布。判定某个样本属于类别1或者0的条件概率如下：

$$P(x)=\frac{e^{\omega\cdot x+b}}{1+e^{\omega\cdot x+b}}$$
$$P(x)=\frac{1}{1+e^{\omega\cdot x+b}}$$

其中$x\in R^n$是输入，$Y\in\{0,1\}$是输出，$\omega\in R^n$称为权值向量，$b\in R$称为偏置。有时为了计算方便，将权值向量和输入向量加以扩充，具体表现为将ω和b连接为$n+1$维向量，而在所有的输入样本x中添加常数1，使其也变为$n+1$维，仍记为ω，x。这样，扩充后的$\omega\cdot x$和原来的$\omega\cdot x+b$表达含义相同。从而，二项逻辑斯蒂回归模型记为：

$$P(x)=\frac{e^{\omega\cdot x}}{1+e^{\omega\cdot x}}=\frac{1}{1+e^{-\omega\cdot x}}=\pi(\omega\cdot x)=z_{\omega}(x)$$
$$P(x)=1-z_{\omega}(x)$$

其中$\pi(\cdot)$就是著名的Sigmoid函数。

对该模型，可以采用极大似然估计法估计模型的参数，似然函数为：

$$\prod_{i=1}^{N}z_{\omega}(x)^{y_i}(1-z_{\omega}(x))^{1-y_i}$$

为了方便求解，通常转换为对数似然函数，并将极大化问题转换为负数的极小化问题：

$$L(\omega)=-\sum_{i=1}^{N}(y_i\log log\, z_{\omega}(x)+(1-y_i)\log log\,(1-z_{\omega}(x)))$$

其中：

$$z_{\omega}(x)=\frac{1}{1+e^{-\omega\cdot x}}$$

就是著名的Sigmoid函数，它有一个非常好的导数性质：

$$z_{\omega}'(x)=z_{\omega}(x)(1-z_{\omega}(x))$$

把$L(\omega)$作为损失函数，采用梯度下降法进行最优化，需要$L(\omega)$对ω求导数，其中ω_j表示参数ω的第j个特征：

$$\frac{\partial L}{\partial \omega_j} = -\sum_{i=1}^{n}\left(y_i \frac{1}{z_\omega(x_i)} - (1-y_i)\frac{1}{1-z_\omega(x_i)}\right)\frac{\partial z_\omega(x_i)}{\partial \omega_j} = -\sum_{i=1}^{n}\left(y_i \frac{1}{z_\omega(x_i)} - (1-y_i)\frac{1}{1-z_\omega(x_i)}\right)z_\omega(x_i)(1-z_\omega(x_i))\frac{\partial \omega \cdot x_i}{\partial \omega_j} = -\sum_{i=1}^{n}($$

```
from functools import reduce
import numpy as np
import pandas as pd

sigmoid = lambda w,x:1/(1+np.exp(-x@w))                    #公式错误!未找到引用源。Sigmoid函数
L = lambda x,y,w:(sigmoid(w,x)-y)@x                        #损失函数的梯度
```

13.5.2 梯度下降法

大多数机器学习或者深度学习算法都涉及某种形式的优化。优化指改变 x 以最小化或最大化某个函数 $f(x)$ 的任务。 我们通常以最小化 $f(x)$ 指代大多数最优化问题。 最大化可经由最小化算法最小化 $-f(x)$ 实现。要最小化或最大化的函数称为目标函数或准则，对其进行最小化时，也称为代价函数、损失函数或误差函数。

通常训练数据为高维数据，直接求解导数等于0的方程比较困难，因此采用梯度下降法进行求解。梯度是最陡峭上升方向, 因此梯度下降沿着梯度的反方向进行权重的更新，可以有效的找到全局的最优解。

算法**13.5.1** 批量梯度下降

输入：训练数据集 $D = \{(x_1, y_1), (x_2, y_2)\ldots, (x_N, y_N)\}$;步长和迭代次数；

输出：最优参数 ω

(1) 对于所有样本的所有特征，按如下公式更新参数：

$$\omega_j = \omega_j - \eta\sum_{i=1}^{n}\left(z_\omega(x_i) - y_i\right)x_i^j \qquad (13.1)$$

(2) 重复步骤(1)，直到参数的每个特征都被更新；

(3) 重复步骤(1)(2)，直到达到指定迭代次数，或算法收敛。

```
def GD(x, y, eta=0.001, iter = 200):                       #梯度下降法
  w = np.zeros(x.shape[1])                                 #初始化w为0
  return reduce(lambda w,_:w-eta*L(x,y,w),range(iter),w)   #更新iter次w
```

批量梯度下降法在每次更新时用所有样本，所有的样本都有贡献，也就是参与调整参数 ω 使其计算得到的是一个标准梯度，对于最优化凸问题，可以达到一个全局最优。理论上一次更新的幅度比较大。如果样本较少，这样收敛的速度会快。但是很多时候，样本数量很大，更新一次要很久, 因此很少直接采用这种方法。

算法**13.5.2**随机梯度下降

(1) 随机打乱样本的顺序；

(2) 对于第i个样本，第j个特征，按照以下公式进行更新：

$$\omega_j = \omega_j - \eta\left(z_\omega(x_i) - y_i\right)x_i^j$$

(3) 重复步骤(2)，直到参数的每个特征都被更新；

(4) 重复步骤(2)(3)，直到每个样本都作用于参数更新；

(5) 重复步骤(2)(3)(4)，直到达到指定迭代次数，或算法收敛。

```
def SGD(x, y, eta=0.001, iter = 200):
        #随机梯度下降
   w = np.zeros(x.shape[1])
        #初始化w为0
   y,x = shuffle(y,x)
                #打乱样本顺序
   update = lambda w,xy: w-eta*L(xy[0].reshape(1,-1),xy[1],w)          #单个样本更新参数
   epoch = lambda w:reduce(update,zip(x,y),w)                              #每轮遍历所
有样本
   return reduce(lambda w,_:epoch(w),range(iter),w)                        #更新iter次
```

随机梯度下降法在每次更新时用单个样本，每次迭代的时候都重新进行随机，参数更新时用样本中的一个例子近似所有样本。因而随机梯度下降可能带来一定的问题，因为计算得到的并不是准确的梯度。对于最优化凸问题，虽然不是每次迭代得到的损失函数都向着全局最优方向，但是大的整体方向是向全局最优解前进，最终的结果往往是在全局最优解附近。但是相比于梯度下降法，这种方法收敛更快，虽然不是全局最优，但很多时候可以接受。

由于随机梯度的权重是基于单个的训练样本进行更新，所以误差曲面也不会像批量梯度下降那么平滑，这也使得随机梯度下降更容易跳出小范围的局部最优解。还可以将随机梯度下降用于在线学习系统中，当有新数据产生时，模型会被实时训练，而不会重置模型的权重。

算法**13.5.3**小批量梯度下降

(1) 随机打乱样本的顺序；

(2) 将样本分为B个批次，对于第k个批次b_k，按照以下公式进行更新：

$$\omega_j = \omega_j - \eta\frac{1}{|b_k|}\sum_{x_i \in b_k}\left(z_\omega(x_i) - y_i\right)x_i^j \qquad (13.2)$$

其中$|b_k|$表示b_k中样本的数量，平均值的目的是因为批次中的样本数可能不同；

(3) 重复步骤(2)，直到每个批次都作用于参数更新；

(4) 重复步骤(2)(3)，直到达到指定迭代次数，或算法收敛。

```
def BGD(x, y, batch,eta=0.001, iter = 200):                              #批量梯度下降
   w = np.zeros(x.shape[1])                                                    #初始化w为0
   y,x = shuffle(y,x)
   bx = np.split(x,range(batch,len(x),batch))                            #分批次
   by = np.split(y.ravel(),range(batch,len(y),batch))                    #分批次
   update = lambda w,xy:w-eta*L(xy[0],xy[1],w)/len(xy)  #单批次更新
```

```
    epoch = lambda w:reduce(update,zip(bx,by),w)                    #每批次更新
    return reduce(lambda w,_:epoch(w),range(iter),w)                #更新iter次w
```

小批量梯度下降是前面两种方法的折中，在实际使用时应用较广。相对与批量梯度下降，它的权重更新比较频繁，收敛速度比较快；相对于随机梯度下降，可以使用向量操作代替循环，从而提高算法的性能。

接下来，以手写数字识别mnist为例，了解以上内容的具体应用。数据集中含有0~9共10个类别，本文将其简化，将非0数字记为1，共分为0和非0两个类别。

```
def loadData(fileName):
    D = pd.read_csv(fileName,header=None,encoding='utf8')
    D['b'] = 1                                                      #w和b合并后
, x添加一维
    return D.iloc[:,1:].values/255,np.array(D[0]==0)                #0和非0两类
if __name__ == '__main__':
    trainData,trainLabel = loadData('./ mnist_train.csv')
    testData,testLabel = loadData('./ mnist_test.csv')
    w = GD(trainData, trainLabel,iter=4)                            #开始训练，学习w
    accuracy = model_test(testData, testLabel, w)                   #验证正确率
    print('the accuracy of GD is:', accuracy)
    w = SGD(trainData, trainLabel,iter=4)                           #开始训练，学习w
    accuracy = model_test(testData, testLabel, w)                   #验证正确率
    print('the accuracy of SGD is:', accuracy)
    w = BGD(trainData, trainLabel,batch=60,iter=4)                  #开始训练，学习w
    accuracy = model_test(testData, testLabel, w)                   #验证正确率
     print('the accuracy of BGD is:', accuracy)
```

```
the accuracy of GD is: 0.9722
the accuracy of SGD is: 0.9917
the accuracy of BGD is: 0.9911
```

13.6 最大熵模型

13.6.1 最大熵原理

最大熵原理是概率模型学习的一个准则。最大熵原理认为，学习概率模型时，在所有可能的概率模型(分布)中，熵最大的模型是最好的模型。通常用约束条件确定概率模型的集合，所以，最大熵原理也可以表述为在满足约束条件的模型集合中选取熵最大的模型。

假设离散随机变量X服从概率分布$P(X)$，则其熵表达为：

$$H(P) = -\sum_{x} P(x) log P(x)$$

其中熵的取值范围是$0 \leq H(P) \leq log|X|$。$|X|$是X的取值个数，当且仅当X的分布是均匀分布时右边等号成立，即均匀分布情况下熵最大。

最大熵原理认为要选择的概率模型首先必须满足已有的事实，即约束条件。在没有更多信息的情况下，那些不确定的部分都是“等可能的”。最大熵原理通过熵的最大化来表示等可能性。“等可能”不容易操作，而熵则是一个可优化的数值指标。

13.6.2 最大熵模型的定义与学习

定义 13.6.1(最大熵模型) 假设满足所有约束条件的模型集合为

$$C \equiv \{P \in P | E_P(f_i) = E_{\tilde{P}}(f_i),\ i = 1, 2, ..., n\}$$（f为特征函数）

定义在条件概率分布$P(X)$上的条件熵为

$$H(P) = -\sum_{x,y} \tilde{P}(x) P(y|x) log P(y|x)$$

则模型集合C中条件熵$H(P)$最大的模型称为最大熵模型。式中的对数为自然对数。

最大熵模型的学习问题可以形式化为约束最优化问题。对于给定的训练数据集 $T = \{(x_1, y_1), (x_2, y_2) ..., (x_N, y_N)\}$, 以及特征函数$f_i(x, y),\ i = 1, 2, ..., n$, 最大熵模型的学习等价于约束最优化问题：

$$H(P) = -\sum_{x,y} \tilde{P}(x) P(x) log P(x)\ s.t. \quad \sum_y P(x) = 1$$
$$E_P(f_i) = E_{\tilde{P}}(f_i),\ i = 1, 2, ..., n$$

引入拉格朗日函数如下

$$L(P, \omega) \equiv -H(P) + \omega_0 \left(1 - \sum_y P(x)\right) + \sum_{i=1}^{n} \omega_i \left(E_P(f_i) = E_{\tilde{P}}(f_i)\right)$$

转换为无约束最优化问题

$$L(P, \omega)$$

进一步转换为易于求解的对偶问题

$$L(P, \omega)$$

设对偶函数$\psi(\omega) = L(P, \omega)$，则最大熵模型的学习可归结为极大化对偶函数的问题。

又可证得条件概率分布的对数似然函数与对偶函数是等价的，故而又可进一步说明最大熵模型的学习问题可归结为极大化对数似然函数的问题。

与13.5章逻辑斯蒂回归模型相比较，可得出二者有类似的形式。最大熵模型和逻辑斯蒂回归模型都是对数线性模型，学习方法都是在给定的训练数据条件下对模型进行极大似然估计或正则化的极大似然估计。

下面介绍几种最大熵模型的学习方法，即改进的迭代尺度法、梯度下降法、牛顿法或拟牛顿法。

算法**13.6.1 (**改进的迭代尺度法**IIS)**

输入：特征函数$f_1, f_2, ..., f_n$;经验分布$\widetilde{P}(X,Y)$, 模型$P_\omega(y|x)$;

输出：最优参数值ω_i^*;最优模型P_{ω^*}。

(1)对所有$i \in \{1,2,...,n\}$, 取初值$\omega_i = 0$。

(2)对每一$i \in \{1,2,...,n\}$

(a)令δ_i是方程

$$\sum_{x,y} \widetilde{P}(x)P(x)f_i(x,y) \exp exp\left(\delta_i f^{\#}(x,y)\right) = E_{\widetilde{P}}(f_i)$$

的解, 这里，

$$f^{\#}(x,y) = \sum_{i=1}^{n} f_i(x,y)$$

(b)更新ω_i值：$\omega_i \leftarrow \omega_i + \delta_i$

(3)如果不是所有ω_i都收敛，重复步（2）。

这一算法关键的一步是步（a），即求解δ_i。如果$f^{\#}(x,y)$是常数, 即对任何x，y, 有$f^{\#}(x,y) = M$, 那么δ_i可以显式地表示为

$$\delta_i = \frac{1}{M} log \frac{E_{\widetilde{P}}(f_i)}{E_P(f_i)}$$

如果$f^{\#}(x,y)$不是常数, 那么必须用数值计算求δ_i。简单有效的方法是牛顿法。以$g(\delta_i) = 0$表示步(a)方程, 牛顿法通过迭代求得δ_i, 使得$g(\delta_i^*) = 0$。迭代公式是

$$\delta_i^{(k+1)} = \delta_i^{(k)} - \frac{g(\delta_i^{(k)})}{g'(\delta_i^{(k)})}$$

只要适当选取初始值$\delta_i^{(0)}$, 由于步(a)的方程有单根，所以牛顿法恒收敛，而且收敛速度很快。

例题:编程实现改进的迭代尺度法IIS。

首先, 定义MaxEntropy类用以实现改进的迭代尺度法。

```
01 import math
02 import collections
03
04 class MaxEntropy():
05     def __init__(self):
06         self._samples = []   #样本集，元素是[y,x1,x2,...]的样本
```

```
07      self._Y = set([]) #标签集合，相当去去重后的y
08      self._numXY = collections.defaultdict(int)  #key为(x,y)，value为出现次数
09      self._N = 0 #样本数
10      self._ep_ = []  #样本分布的特征期望值
11      self._xyID = {}  #key记录(x,y),value记录id号
12      self._n = 0 #特征的个数
13      self._C = 0  #最大特征数
14      self._IDxy = {}   #key为(x,y)，value为对应的id号
15      self._w = []
16      self._EPS = 0.005  #收敛条件
17      self._lastw = []   #上一次w参数值
18
19  def loadData(self,filename):
20      with open(filename) as fp:
21          self._samples = [item.strip().split('\t') for item in fp.readlines()]
22      for items in self._samples:
23              y = items[0]
24              X = items[1:]
25              self._Y.add(y)
26              for x in X:
27                  self._numXY[(x,y)] += 1
28
29  def _sample_ep(self):  #计算特征函数fi关于经验分布的期望
30      self._ep_ = [0] * self._n
31      for i,xy in enumerate(self._numXY):
32          self._ep_[i] = self._numXY[xy]/self._N
33          self._xyID[xy] = i
34          self._IDxy[i] = xy
35
36  def _initparams(self): #初始化参数
37      self._N = len(self._samples) #14
38      self._n = len(self._numXY) # 19
39      self._C = max([len(sample)-1 for sample in self._samples]) # 4
40      self._w = [0]*self._n # [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
41      self._lastw = self._w[:]
42      self._sample_ep()               #计算每个特征关于经验分布的期望
43
44  def _Zx(self,X):   #计算每个x的Z值
45      zx = 0
46      for y in self._Y:
47          ss = 0
48          for x in X:
49              if (x,y) in self._numXY:
50                  ss += self._w[self._xyID[(x,y)]]
51          zx += math.exp(ss)
52      return zx
53
54  def _model_pyx(self,y,X):  #计算每个P(y|x)
55      Z = self._Zx(X)
56      ss = 0
```

```
        for x in X:
            if (x,y) in self._numXY:
                ss += self._w[self._xyID[(x,y)]]
        pyx = math.exp(ss)/Z
        return pyx

    def _model_ep(self,index):  #计算特征函数fi关于模型的期望
        x,y = self._IDxy[index]
        ep = 0
        for sample in self._samples:
            if x not in sample:
                continue
            pyx = self._model_pyx(y,sample)
            ep += pyx/self._N
        return ep

    def _convergence(self):
        for last,now in zip(self._lastw,self._w):
            if abs(last - now) >=self._EPS:
                return False
        return True

    def predict(self,X):  #计算预测概率
        Z = self._Zx(X)
        result = {}
        for y in self._Y:
            ss = 0
            for x in X:
                if (x,y) in self._numXY:
                    ss += self._w[self._xyID[(x,y)]]
            pyx = math.exp(ss)/Z
            result[y] = pyx
        return result

    def train(self,maxiter = 1000):  #训练数据
        self._initparams()
        for _ in range(0,maxiter):  #最大训练次数
            self._lastw = self._w[:]
            for i in range(self._n):
                ep = self._model_ep(i)   #计算第i个特征的模型期望
                self._w[i] += math.log(self._ep_[i]/ep)/self._C  #更新参数
            if self._convergence():  #判断是否收敛
                break
```

读取数据，实例化MaxEntropy类用以实现改进的迭代尺度法。

```
01 maxent = MaxEntropy()
02 x = ['sunny','hot','high','FALSE'] # 分类结果应为“no”
03 maxent.loadData('dataset.txt')
04 maxent.train()
```

```
05 print('predict :',maxent.predict(x)) # 输出分类结果预测概率
01 predict :{'no': 0.9994803665107912, 'yes': 0.0005196334892088451}
```

算法**13.6.2 (**最大熵模型学习的**BFGS**算法**)**

输入：特征函数$f_1, f_2, ..., f_n$;经验分布$\widetilde{P}(X,Y)$, 目标函数$f(\omega)$, 梯度$g(\omega) = \nabla f(\omega)$, 精度要求$\varepsilon$;

输出：最优参数值ω^*;最优模型P_{ω^*}

(1)选定初始点$\omega^{(0)}$, 取B_0为正定对称矩阵，置$k = 0$;

(2)计算$g_k = g\left(\omega^{(k)}\right)$。若$\|g_k\| < \varepsilon$, 则停止计算，得$\omega^* = \omega^{(k)}$;否则转（3）；

(3)由$B_k p_k = -g_k$求出p_k;

(4)一维搜索：求λ_k使得

$$f\left(\omega^{(k)} + \lambda_k p_k\right) = f(\omega^{(k)} + \lambda p_k)$$

(5)置$\omega^{(k+1)} = \omega^{(k)} + \lambda_k p_k$;

(6)计算$g_{k+1} = g\left(\omega^{(k+1)}\right)$, 若$\|g_{k+1}\| < \varepsilon$, 则停止计算，得$\omega^* = \omega^{(k+1)}$;否则，按下式求出$B_{k+1}$:

$$B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T \delta_k} - \frac{B_k \delta_k \delta_k^T B_k}{\delta_k^T B_k \delta_k}$$

其中,

$$y_k = g_{k+1} - g_k,\ \delta_k = \omega^{(k+1)} - \omega^{(k)}$$

(7)置$k = k + 1$, 转（3）。

13.7 SVM

与感知机模型相似, 支持向量机模型(SVM)是一个经典的二分类模型。该模型本质上是定义在特征空间上的间隔最大的线性分类器。其中，间隔最大使其区别于感知机模型，通过间隔最大化的学习策略，SVM可转换为求解一个凸二次规化问题的最优解的过程。

支持向量机可解决线性可分问题、线性不可分问题及非线性问题，后面将依次给出使用SVM解决这三类问题的过程。

13.7.1 线性可分支持向量机与硬间隔最大化

定义 **13.7.1(**线性可分支持向量机**)** 给定线性可分训练数据集, 通过间隔最大化或等价地求解相应的凸二次规划问题学习得到的分离超平面为

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

以及相应的分类决策函数

$$f(x) = sign(\omega^{*} \cdot x + b^{*})$$

称为线性可分支持向量机。可以证明最大间隔分离超平面具有存在唯一性。

下面再介绍两个关键概念：函数间隔和几何间隔。

定义**13.7.2** **(**函数间隔**)** 对于给定的训练数据集T和超平面(ω,b), 定义超平面(ω,b)关于样本点(x_i, y_i)的函数间隔为

$$\hat{\gamma}_i = y_i\left(\omega \cdot x_i + b\right)$$

定义超平面(ω,b)关于训练数据集T的函数间隔为超平面(ω,b)关于T中所有样本点(x_i, y_i)的函数间隔之最小值，即

$$\hat{\gamma} = \hat{\gamma}_i$$

但函数间隔不能够准确地定量衡量分类准确性，因为对于同一分离超平面$\omega \cdot x + b = 0$, 若同时将权重ω和偏置b乘上一个相同的系数a, 该分离超平面不发生改变，但函数间隔变为原来的a倍, 故而函数间隔只能定性衡量分类准确性。因此需要引入几何间隔定量衡量分类准确性。

定义**13.7.3** **(**几何间隔**)** 对于给定的训练数据集T和超平面(ω,b), 定义超平面(ω,b)关于样本点(x_i, y_i)的几何间隔为：

$$\gamma_i = \frac{\hat{\gamma}_i}{\|\omega\|} = y_i\left(\frac{\omega}{\|\omega\|} \cdot x_i + \frac{b}{\|\omega\|}\right)$$

定义超平面(ω,b)关于训练数据集T的函数间隔为超平面(ω,b)关于T中所有样本点(x_i, y_i)的几何间隔之最小值，即

$$\gamma = \gamma_i$$

由于几何间隔考虑到通过将函数间隔除以$\|\omega\|$用以排除权重ω和偏置b乘上相同系数带来的变化，又可以通过几何证明得到几何间隔确为实例点与分离超平面几何上的距离，故而几何距离具有唯一确定性, 能够很好地定量评估分类准确性。

此外, 函数间隔$\hat{\gamma}_i$的取值并不影响最优化问题的解，通常取$\hat{\gamma} = 1$。此时，在线性可分情况下, 满足$\hat{\gamma} = 1$条件的样本点与分离超平面距离最近，把这些样本点的实例称为支持向量, 即满足：

$$y_i\left(\omega \cdot x_i + b\right) - 1 = 0$$

如图13.7.1所示, 在H_1和H_2上的点就是支持向量。其中H1: $\omega \cdot x + b = 1$且H2: $\omega \cdot x + b = -1$。H_1和H_2称为间隔边界。H1和H_2之间的距离称为间隔。间隔依赖于分离超平面的法向量ω, 等于$\frac{2}{\|\omega\|}$。对于$y_i = +1$的正例点在H1上, 对于$y_i = -1$的正例点在H2上。只有支持向量决定分离超平面, 而支持向量的数目一般很少, 所以支持向量机由很少的“重要的”训练样本决定。

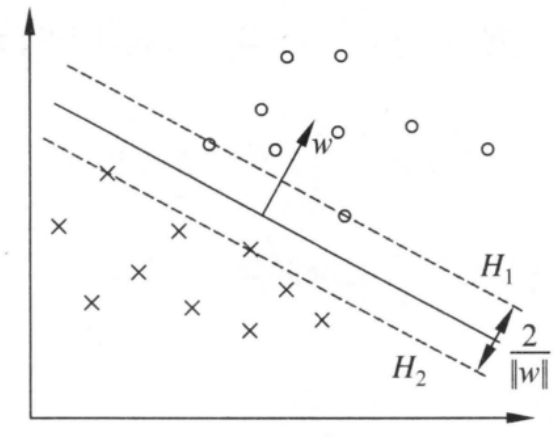


图13. 7. 1 支持向量

对于线性可分二分类问题，可以通过间隔最大化方法进行学习。$\frac{2}{\|\omega\|}$的最大化等价于$\frac{1}{2}\|\omega\|^2$的最小化。

算法**13.7.1 (**线性可分支持向量机学习算法——最大间隔法**)**

输入：线性可分训练数据集$T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$, 其中,

$x_i \in X = R^n, y_i \in Y = \{-1, +1\}, i = 1, 2, ..., N$;

输出：最大间隔分离超平面和分类决策函数。

构造并求解约束最优化问题：

$$\frac{1}{2}\|\omega\|^2 \; s.t. \quad 1 - y_i(\omega \cdot x_i + b) \leq 0, \quad i = 1, 2, ..., N$$

(1) 求得最优解ω^*, b^*

(2) 由此得到分离超平面:

$$\omega^* \cdot x + b^* = 0$$

分类决策函数

$$f(x) = sign(\omega^* \cdot x + b^*)$$

例题:有实例点(3,3)，(4,3)，(1,1)分别对应标签值为1，1，-1，请使用最大间隔法求解SVM类函数$f(x) = sign(\omega \cdot x + b)$。在二维空间中，$\omega = \{\omega_1, \omega_2\}$且$x = \{x_1, x_2\}$。

解:**所求最优化问题如下**：

$$\begin{aligned}\min_{w,b} \quad & \frac{1}{2}(w_1^2+w_2^2)\\ \text{s.t.} \quad & 3w_1+3w_2+b\geqslant 1\\ & 4w_1+3w_2+b\geqslant 1\\ & -w_1-w_2-b\geqslant 1\end{aligned}$$

观察可知该问题为约束非线性规划问题。

构造拉格朗日函数

$$L(\omega,\lambda)=\frac{1}{2}\omega_1^2+\frac{1}{2}\omega_2^2-\lambda_1\left(3\omega_1+3\omega_2+b-1\right)-\lambda_2\left(4\omega_1+3\omega_2+b-1\right)-\lambda_3\left(-\omega_1-\omega_2-b-1\right)$$

令$\nabla_\omega L(\omega,\lambda)=0$，可得：

$$\left(\omega_1\ \omega_2\ 0\right)-\lambda_1(3\ 3\ 1)-\lambda_2(4\ 3\ 1)-\lambda_3(-1\ -1\ -1)=0$$

故而需求解以下方程组：

$$\{\omega_1-3\lambda_1-4\lambda_2+\lambda_3=0 \quad (1)\ \omega_2-3\lambda_1-3\lambda_2+\lambda_3=0 \quad (2)-\lambda_1-\lambda_2+\lambda_3=0 \quad (3)\ \lambda_1\left(3\omega_1+3\omega\right.$$

可解得：

$$\{\lambda_1=\lambda_3=\frac{1}{4}\ \lambda_2=0\ \omega_1=\omega_2=\frac{1}{2}\ b=-2$$

由于$\lambda_1,\lambda_2,\lambda_3\geq 0$，满足K-T条件，并且所得结果为唯一K-T点，故而所得结果为最优点，即为该最优化问题的解，其中实例点(3,3)，(1,1)为支持向量。

此外，还可参照13.1节感知机算法梯度下降算法求解支持向量机。

算法**13.7.2 (**线性可分支持向量机学习算法——梯度下降法**)**

输入：训练数据集$(x_1, y_1), (x_2, y_2)\ldots, (x_N, y_N)\}$，其中$x_i\in R^n$，$y_i\in\{-1,+1\}$，$i=1, 2, .., N$；学习率$\eta(0<\eta\leq 1)$；

输出：ω, b；感知机模型$f(x)=sign(\omega\cdot x_i+b)$。

(1) 选取初值ω_0, b_0；

(2) 在训练集中选取数据(x_i, y_i)；

(3) 如果$y_i(\omega\cdot x_i+b)\leq\|\omega\|$，

$$\omega\leftarrow\omega+\eta y_i x_i \qquad b\leftarrow b+\eta y_i$$

(4) 转至(2)，直至训练集中没有误分类点。

例题：编程实现SVM算法的原始形式，求模型$f(x)=sign(\omega\cdot x+b)$。

首先，引入基本的库函数并进行画图参数配置。

```
01 import numpy as np
02 import pandas as pd
03 import matplotlib.pyplot as plt
```

```
04 #画图参数配置
05 plt.rcParams['font.family'] = ['sans-serif']
06 plt.rcParams['font.sans-serif'] = ['SimHei']
07 plt.rcParams['axes.unicode_minus']=False
08 plt.axes().set_aspect('equal')                                    # 使x,y轴的单位长度一致
```

构造具体实现过程。

```
01 def svm(data,w,b,lr):                                            # 数据, 参数, 偏置, 学习率
02     X=data[:,:-1]
03     Y=data[:,-1]
04     count=0
                                # 迭代次数
05     ret = [[0,None,0,0,0]]
06     error_point = not None
07     while error_point:
08         hyperplane(count,X,Y,w,b)
09         error_point=None
10         for i in range(len(X)):
11             if Y[i]*(np.dot(w,X[i])+b)<= (w[0]**2+w[1]**2)**0.5:   # 误分类
12                 w=w+lr*Y[i]*X[i]                                   # 更新参数
13                 b=b+lr*Y[i]                                        # 更新偏置
14                 error_point = f'x{i+1}'                            # 记录误分类点
15                 break
16         count+=1
17         if(error_point): ret.append([count,error_point,w,b,f'{w[0]}*x1+{w[1]}*x2+{b}'])
18     return  ret
```

为了直观的展现超平面的效果，采用hyperplane函数进行绘制。

```
01 def hyperplane(index,X,Y,w,b):
02     positive = X[np.where(Y==1)]
         # 正样本
03     plt.plot(positive[:,0], positive[:,1],'ro')
         # 画正样本
04     negative = X[np.where(Y==-1)]
         # 负样本
05     plt.plot(negative[:,0], negative[:,1],'bx')
         # 画负样本
06     xlist = np.linspace(0.0, 5.0, 100)
07     y_line = [float('inf')]*len(xlist) if w[1]==0 else (-w[0]*xlist-b)/w[1]
08     plt.plot(xlist, y_line, color='k', linewidth=1, linestyle='-')   # 绘制超平面直线
09     plt.axis([0.0, 5.0, 0.0, 5.0]) # 坐标轴显示范围
10     plt.title(f'第{index}次迭代结果')
11     plt.show()
```

最后，读取数据，调用以上函数完成SVM算法的分类。

```
01 data=pd.read_csv('example1.1.csv',header=None,encoding='utf8').values
02 w,b,lr=np.array([0,0]),0,1    #参数初始化
03 ret=svm(data,w,b,lr)
04 df = pd.DataFrame(ret,columns=['迭代次数','误分类点','w','b','模型'])
05 print(df)
```

	迭代次数	误分类点	w	b	模型
0	0	None	0	0	0
1	1	x1	[-1, -1]	-1	-1*x1+-1*x2+-1
2	2	x2	[2, 2]	0	2*x1+2*x2+0
3	3	x1	[1, 1]	-1	1*x1+1*x2+-1
4	4	x1	[0, 0]	-2	0*x1+0*x2+-2
5	5	x2	[3, 3]	-1	3*x1+3*x2+-1
6	6	x1	[2, 2]	-2	2*x1+2*x2+-2
7	7	x1	[1, 1]	-3	1*x1+1*x2+-3
8	8	x1	[0, 0]	-4	0*x1+0*x2+-4
9	9	x2	[3, 3]	-3	3*x1+3*x2+-3
10	10	x1	[2, 2]	-4	2*x1+2*x2+-4
11	11	x1	[1, 1]	-5	1*x1+1*x2+-5
12	12	x2	[4, 4]	-4	4*x1+4*x2+-4
13	13	x1	[3, 3]	-5	3*x1+3*x2+-5
14	14	x1	[2, 2]	-6	2*x1+2*x2+-6
15	15	x1	[1, 1]	-7	1*x1+1*x2+-7
16	16	x2	[4, 4]	-6	4*x1+4*x2+-6
17	17	x1	[3, 3]	-7	3*x1+3*x2+-7
18	18	x1	[2, 2]	-8	2*x1+2*x2+-8

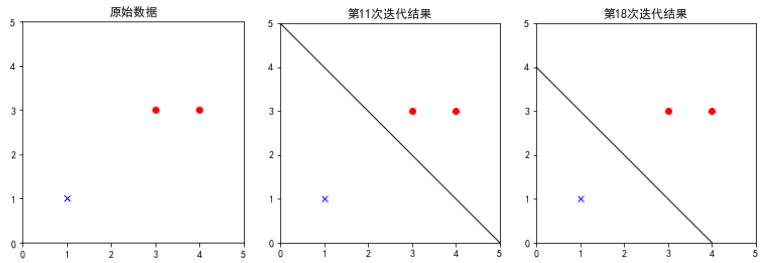


图13.7.2 SVM超平面绘制结果(正实例点是$x_1=(3,3)^T$, $x_2=(4,3)^T$, 负实例点是$x_3=(1,1)^T$)

回顾图13.2，可以发现SVM与感知机在相同训练任务上有两个较为明显的差异：

1　感知机模型迭代了7次，而SVM迭代了18次

2　感知机最终模型只是实现了正负样本点的分离，而SVM在此基础上还实现了间隔最大化

由以上两点可以得出SVM的学习过程相较于感知机更为严格，这在后面要介绍的合页损失函数中还会补充说明。

此外，可回顾13.1节感知机内容，可通过求解问题的对偶问题来得到原始问题的解，即SVM的对偶算法。

这样做有三个优点：

1　对偶问题往往更易求解

2　自然引入核函数, 从而可推广至非线性问题的求解

3　引入Gram矩阵减少运算量

算法**13.7.3 (**线性可分支持向量机学习算法——对偶形式**)**

输入：线性可分训练数据集$\{(x_1, y_1), (x_2, y_2)\dots, (x_N, y_N)\}$, 其中

$x_i \in R^n$, $y_i \in \{-1, +1\}$, $i = 1, 2, .., N$;

输出：分离超平面和分类决策函数

(1)构造并求解约束最优化问题

$$\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j (x_i \cdot x_j) - \sum_{i=1}^{N}\alpha_i \; s.t. \quad \sum_{i=1}^{N}\alpha_i y_i = 0 \alpha_i \geq 0, \; i = 1, 2, \; \dots, N$$

求得最优解$\alpha^* = (\alpha_1^*, \alpha_2^*, \; \dots, \alpha_N^*)^T$

(2)计算

$$\omega^* = \sum_{i=1}^{N}\alpha_i^* y_i x_i$$

并选择α^*的一个正分量$\alpha_j^* > 0$, 计算

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

(3)求得分离超平面

$$\omega^* \cdot x + b^* = 0$$

分类决策函数：

$$f(x) = sign(\omega^* \cdot x + b^*)$$

根据以上算法可总结出，对于给定的线性可分训练数据集，可以首先求解对偶问题的解α^*, 再去求原始问题的解ω^*和b^*, 从而求取分离超平面和分类决策函数。该算法是线性可分支持向量机学习的基本算法。

注意, 该算法中选择的$\alpha_j^* > 0$对应的样本点是支持向量，而其他样本点对求解ω^*和b^*没有影响。

例题:有实例点(3,3)，(4,3)，(1,1)分别对应标签值为1，1，-1，请使用线性可分支持向量机对偶学习算法求解SVM分类函数$f(x) = sign(\omega \cdot x + b)$

解:构造该问题的对偶问题

$\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j\left(x_i\cdot x_j\right) - \sum_{i=1}^{N}\alpha_i = \frac{1}{2}\left(18\alpha_1^2 + 25\alpha_2^2 + 2\alpha_3^2 + 42\alpha_1\alpha_2 - 12\alpha_1\alpha_3 - 14\alpha_2\alpha_3\right) - \alpha_1 - \alpha_2 - \alpha_3 s.t. \quad \alpha_1 +$

将$\alpha_3 = \alpha_1 + \alpha_2$代入目标函数并将目标函数记为

$$s\left(\alpha_1, \alpha_2\right) = 4\alpha_1^2 + \frac{13}{2}\alpha_2^2 + 10\alpha_1\alpha_2 - 2\alpha_1 - 2\alpha_2$$

分别求s关于α_1和α_2的偏导数取零，可得s在$\left(\frac{3}{2}, -1\right)^T$处取得极值，但该点不满足$\alpha_2 \geq 0$，故而极值应在边界处（坐标轴上）取得。当$\alpha_1 = 0$时，最小值$s\left(0, \frac{3}{2}\right) = -\frac{2}{13}$；当$\alpha_2 = 0$时，最小值$s\left(\frac{1}{4}, 0\right) = -\frac{1}{4}$。由于后者更小，所以最终解$\alpha^* = \left(\frac{1}{4}, 0, \frac{1}{4}\right)^T$。取其中一正分量$\alpha_1 = \frac{1}{4}$，计算得$\omega_1^* = \omega_2^* = \frac{1}{2}$，$b^* = -2$。

求得分离超平面为 $\frac{1}{2}x^{(1)} + \frac{1}{2}x^{(2)} - 2 = 0$

分类决策函数为 $f(x) = sign(\frac{1}{2}x^{(1)} + \frac{1}{2}x^{(2)} - 2)$

算法**13.7.4 (**线性可分支持向量机学习算法——梯度下降法对偶形式**)**

输入：训练数据集$\left\{\left(x_1, y_1\right), \left(x_2, y_2\right)\ldots, \left(x_N, y_N\right)\right\}$, 其中$x_i \in R^n$, $y_i \in \{-1, +1\}$, $i = 1, 2, .., N$;学习率$\eta(0 < \eta \leq 1)$;

输出：α, b; SVM模型$f(x) = sign\left(\sum_{j=1}^{N}\alpha_j y_j x_j \cdot x + b\right)$, $\alpha = \left(\alpha_1, \alpha_2, \ldots\alpha_N\right)^T$

(1) $\alpha \leftarrow 0$, $b \leftarrow 0$;

(2) 在训练集中选取数据$\left(x_i, y_i\right)$;

(3) 如果$y_i\left(\sum_{j=1}^{N}\alpha_j y_j x_j \cdot x_i + b\right) \leq \|\omega\| = \sum_{j=1}^{N}\alpha_j y_j x_j$,

$$\alpha_i \leftarrow \alpha_i + \eta \quad b \leftarrow b + \eta y_i$$

(4) 转至(2)，直至训练集中没有误分类点。

例题：编程实现SVM算法的对偶形式求SVM模型。

```
01 def dual_SVM(data,alpha,b,lr):                                  # 数据，alpha，偏置，学习率
02     X = data[:,:-1]
03     Y = data[:,-1]
04     count = 0
05     w= [0, 0]
06     ret = [[0,None,alpha.copy(),0]]
07     error_point = not None
08     Gram_matrix = X@X.T                                         # 构造Gram矩阵
09     w = ((alpha * Y)[:, np.newaxis] * X).sum(0)
```

```
    while error_point:
        hyperplane(count,X,Y,w,b)
        error_point = None
        for i in range(len(Y)):
            if Y[i]*((alpha*Y*Gram_matrix[i]).sum() + b) <= (w[0]**2+w[1]**2)**0.5:
                alpha[i] += lr                                          # 更新alpha
                b += Y[i]                                               # 更新偏置
                error_point = f'x{i+1}'                                 # 记录误分类点
                break
        count += 1
        w = ((alpha*Y)[:,np.newaxis]*X).sum(0)
        if(error_point):
            ret.append([count,error_point,alpha.copy(),b,w,f'{w[0]}*x1+{w[1]}*x2+{b}'])
    return ret
```

最后，读取数据。

```
data=pd.read_csv('example1.1.csv',header=None,encoding='utf8').values
alpha,b,lr = np.zeros(3), 0, 1
ret = dual_SVM(data,alpha,b,lr)
df = pd.DataFrame(ret,columns=['迭代次数','误分类点','α','b','w','模型'])
print(df)
```

	迭代次数	误分类点	α	b	w	模型
0	0	None	[0.0, 0.0, 0.0]	0	None	None
1	1	x1	[1.0, 0.0, 0.0]	-1	[-1.0, -1.0]	-1.0*x1+-1.0*x2+-1
2	2	x2	[1.0, 1.0, 0.0]	0	[2.0, 2.0]	2.0*x1+2.0*x2+0
3	3	x1	[2.0, 1.0, 0.0]	-1	[1.0, 1.0]	1.0*x1+1.0*x2+-1
4	4	x1	[3.0, 1.0, 0.0]	-2	[0.0, 0.0]	0.0*x1+0.0*x2+-2
5	5	x2	[3.0, 2.0, 0.0]	-1	[3.0, 3.0]	3.0*x1+3.0*x2+-1
6	6	x1	[4.0, 2.0, 0.0]	-2	[2.0, 2.0]	2.0*x1+2.0*x2+-2
7	7	x1	[5.0, 2.0, 0.0]	-3	[1.0, 1.0]	1.0*x1+1.0*x2+-3
8	8	x1	[6.0, 2.0, 0.0]	-4	[0.0, 0.0]	0.0*x1+0.0*x2+-4
9	9	x2	[6.0, 3.0, 0.0]	-3	[3.0, 3.0]	3.0*x1+3.0*x2+-3
10	10	x1	[7.0, 3.0, 0.0]	-4	[2.0, 2.0]	2.0*x1+2.0*x2+-4
11	11	x1	[8.0, 3.0, 0.0]	-5	[1.0, 1.0]	1.0*x1+1.0*x2+-5
12	12	x2	[8.0, 4.0, 0.0]	-4	[4.0, 4.0]	4.0*x1+4.0*x2+-4
13	13	x1	[9.0, 4.0, 0.0]	-5	[3.0, 3.0]	3.0*x1+3.0*x2+-5
14	14	x1	[10.0, 4.0, 0.0]	-6	[2.0, 2.0]	2.0*x1+2.0*x2+-6
15	15	x1	[11.0, 4.0, 0.0]	-7	[1.0, 1.0]	1.0*x1+1.0*x2+-7
16	16	x2	[11.0, 5.0, 0.0]	-6	[4.0, 4.0]	4.0*x1+4.0*x2+-6
17	17	x1	[12.0, 5.0, 0.0]	-7	[3.0, 3.0]	3.0*x1+3.0*x2+-7
18	18	x1	[13.0, 5.0, 0.0]	-8	[2.0, 2.0]	2.0*x1+2.0*x2+-8

以上即为线性可分支持向量机的学习算法。但在实际处理的问题中，可能会有一些特异点导致使用硬间隔最大化方法无法进行求解。因此，我们在后面引入更加一般化的软间隔最大化方法来解决这一类问题。

13.7.2 线性支持向量机与软间隔最大化

针对有一些特异点存在的样本分布，可能引入复杂的超曲面也可以达到很好的分类效果，

但是绝大多数样本点都可以用超平面分离的情况下，线性超平面仍然是最优模型。

所以, 我们放宽SVM的限制，可以允许存在一些误分类的点。间隔化“硬”为“软”，这便是软间隔最大化的思想。

具体描述为：线性不可分线性不可分意味着某些样本点(x_i, y_i)不能满足函数间隔大于等于1的约束条件，为了解决这个问题，可以对每个样本点(x_i, y_i)引进一个松弛变量$\zeta_i \geq 0$, 使得函数间隔加上松弛变量大于等于1.这样约束条件变为

$$y_i\left(\omega \cdot x_i + b\right) \geq 1 - \zeta_i$$

松弛变量的引入就是为了对付那些特异点的，其他点（姑且称作正常点）的松弛变量都是0。

同时，对每个松弛变量ζ_i, 支付一个代价ζ_i。目标函数由原来的$\frac{1}{2}\|\omega\|^2$变成

$$\frac{1}{2}\|\omega\|^2 + C\sum_{i=1}^{N}\zeta_i$$

其中, $C > 0$称为惩罚参数，用于调和两项之间的重要关系。最小化目标函数包含两层含义:使得$\frac{1}{2}\|\omega\|^2$尽量小即间隔尽量大，同时使得误分类点的个数尽量小。

线性不可分的线性支持向量机的学习问题变成如下凸二次规划问题（原始问题）：

$$\frac{1}{2}\|\omega\|^2 + C\sum_{i=1}^{N}\zeta_i s.t. \quad y_i\left(\omega \cdot x_i + b\right) \geq 1 - \zeta_i, \quad i = 1, 2, ..., N$$
$$\zeta_i \geq 0, \quad i = 1, 2, ..., N$$

由于原始问题是凸二次规划问题，故而关于(ω, b, ζ)的解是存在的。可以证得ω的解是唯一的, 但b的解可能不唯一, 而是存在于一个区间。

下面给出线性支持向量机的定义。

定义 **13.7.4(**线性支持向量机**)** 给定线性不可分训练数据集, 通过求解相应的凸二次规划问题学习得到的分离超平面为

$$\omega^* \cdot x + b^* = 0$$

以及相应的分类决策函数

$$f(x) = sign(\omega^* \cdot x + b^*)$$

称为线性支持向量机。

相应的，该问题也存在对偶问题，进而介绍线性支持向量机学习算法。

算法**13.7.4 (**线性支持向量机学习算法**)**

输入：训练数据集$T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$, 其中,

$x_i \in X = R^n, y_i \in Y = \{-1, +1\}, i = 1, 2, ..., N$;

输出：最大间隔分离超平面和分类决策函数。

(1) 选择惩罚参数$C > 0$，构造并求解约束最优化问题：

$$\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j\left(x_i \cdot x_j\right) - \sum_{i=1}^{N}\alpha_i s.t. \quad \sum_{i=1}^{N}\alpha_i y_i = 0 \; 0 \le \alpha_i \le C, i = 1,2, \ \ldots, N$$

求得最优解$\alpha^* = (\alpha_1^*, \alpha_2^*, \ \ldots, \alpha_N^*)^T$

(2)计算

$$\omega^* = \sum_{i=1}^{N}\alpha_i^* y_i x_i$$

并选择α^*的一个正分量$0 < \alpha_j^* < C$，计算

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

(3)求得分离超平面

$$\omega^* \cdot x + b^* = 0$$

分类决策函数：

$$f(x) = sign(\omega^* \cdot x + b^*)$$

步骤（2）中，对任一适合$0 < \alpha_j^* < C$的α_j^*都可以求出b^*，从理论上讲，b的解可能不唯一，然而在实际应用中，往往只会出现算法叙述的情况。

此外，讨论软间隔的支持向量。软间隔的支持向量x_i或者在间隔边界上，或者在间隔边界与分离超平面之间，或者在分离超平面误分一侧。若$\alpha_i^* < C$，则$\zeta_i = 0$，支持向量x_i恰好落在间隔边界上；若$\alpha_i^* = C, 0 < \zeta_i < 1$，则分类正确，$x_i$在间隔边界与分离超平面之间；若$\alpha_i^* = C, \zeta_i = 1$，则$x_i$在分离超平面上；若$\alpha_i^* = C, \zeta_i > 1$，则$x_i$在分离超平面误分一侧。

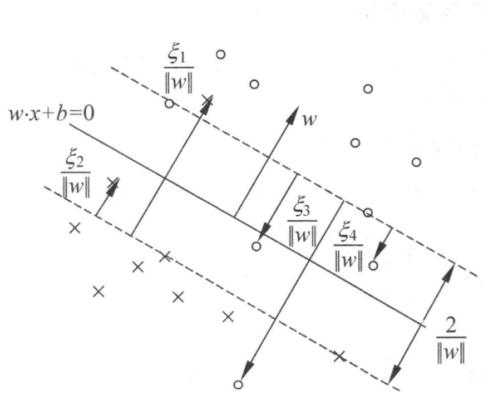


图13.7.2 软间隔的支持向量

对于线性支持向量机还有以下解释，即最小化以下目标函数：

$$\sum_{i=1}^{N}[1-y_i(\omega\cdot x_i+b)]_+ + \lambda\|\omega\|^2$$

其中第2项为系数为λ的ω的L2范数，为正则化项。第1项为经验损失，称为合页损失函数。下标“+”表示以下取正值的函数。

$$[z]_+ = \{z,\ z>0\ 0,\ z\le 0$$

合页损失函数图像如下：

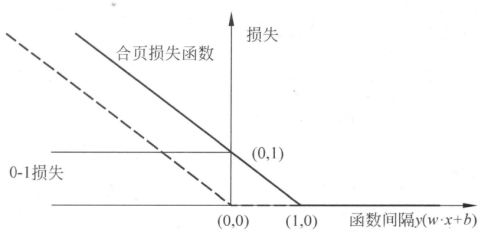


图13.7.3 合页损失函数

综合合页损失函数的表达式和图像可以得出，当样本点被正确分类且函数间隔（确信度）$y_i(\omega\cdot x_i+b)$大于等于1时，损失才为0，否则损失是$1-y_i(\omega\cdot x_i+b)$；相比之下，图中虚线所示为感知机的损失函数，当样本点被正确分类即可认为损失为0，否则损失为$-y_i(\omega\cdot x_i+b)$。

由此可得，合页损失函数不仅要求分类正确，而且确信度足够高时损失才是0。也就是说，合页损失函数对学习有更高的要求。

13.7.3 非线性支持向量机与核函数

当分类问题是非线性问题时，可以通过利用核技巧，使用非线性支持向量机进行求解。

见如下例子：如图13.7.4左图，是一个分类问题，图中“· ”表示正实例点，“×”表示负实例点。由图可见，无法用直线（线性模型）将正负实例正确分开，但可以用一条椭圆曲线（非线性模型）将它们正确分开。一般来说，对于给定的训练数据集，如果能R^n中的一个超曲面将正负实例正确分类开，则称该问题为非线性可分问题。

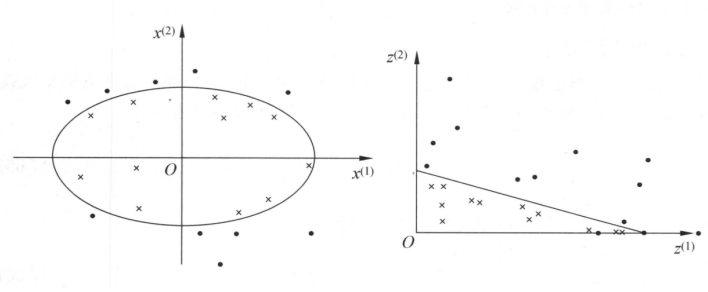


图13.7.4 非线性分类问题与核技巧示例

非线性问题往往不好求解，所以希望能用解线性分类问题的方法解决这个问题。所采取的

方法是进行一个非线性变换，将非线性问题变换为线性问题，通过解变换后的线性问题的方法求解原来的非线性问题。对图13.7.4所示的例子，通过变换，将左图中椭圆变换成右图中的直线，将非线性分类问题变换为线性分类问题。

设原空间为$X \subset R^2$, $x = (x^{(1)}, x^{(2)})^T \in X$, 新空间为$Z \subset R^2$, $z = (z^{(1)}, z^{(2)})^T \in Z$, 定义从原空间到新空间的变换（映射）

$$z = \varphi(x) = \left(\left(x^{(1)}\right)^2, \left(x^{(2)}\right)^2 \right)^T$$

经过变换$z = \varphi(x)$, 原空间$X \subset R^2$中的椭圆

$$\omega_1 \left(x^{(1)}\right)^2 + \omega_2 \left(x^{(2)}\right)^2 + b = 0$$

变换为新空间$Z \subset R^2$中的直线

$$\omega_1 z^{(1)} + \omega_2 z^{(2)} + b = 0$$

这样，原空间的非线性可分问题就变成了新空间的线性可分问题。

上面的例子说明，用线性分类方法求解非线性分类问题分为两步：首先使用一个变换将原空间的数据映射到新空间；然后在新空间里用线性分类学习方法从训练数据中学习分类模型。核技巧就属于这样的方法。

核技巧应用到支持向量机，其基本想法就是通过一个非线性变换将输入空间（欧氏空间R^n或离散集合）对应于一个特征空间（希尔伯特空间H），使得在输入空间R^n中的超曲面模型对应于特征空间H中的超平面模型（支持向量机）。这样，分类问题的学习任务通过在特征空间中求解线性支持向量机就可以完成。

下面介绍核函数的定义。

定义 **13.7.5(**核函数**)** 设X是输入空间（欧氏空间R^n的子集或离散集合），又设H是特征空间（希尔伯特空间H），如果存在一个从X到H的映射

$$\varphi(x): X \to H$$

使得对所有$x, z \in X$, 函数$K(x, z)$满足条件

$$K(x, z) = \varphi(x) \cdot \varphi(z)$$

则称$K(x, z)$为核函数，$\varphi(x)$为映射函数，式中$\varphi(x) \cdot \varphi(z)$为两者内积。

核技巧的想法是，在学习与预测中只定义核函数$K(x, z)$, 而不显式地定义映射函数φ。通常，直接计算$K(x, z)$比较容易，而通过$\varphi(x)$和$\varphi(x)$计算$K(x, z)$并不容易。注意，φ是输入空间R^n到特征空间H的映射，特征空间H一般是高维的，甚至是无穷维的。可以看到，对于给定的核$K(x, z)$，特征空间H和映射函数φ的取法并不唯一，可以取不同的特征空间，即便是在同一特征空间里也可以取不同的映射。

常用的核函数有多项式核函数、高斯核函数和字符串核函数。

再关注到求解非线性问题的支持向量机的对偶问题。

定义**13.7.6(**非线性支持向量机**)** 从非线性分类训练集, 通过核函数与软间隔最大化, 学习得到的分类决策函数

$$f(x) = sign\left(\sum_{i=1}^{N} \alpha_i^* y_i K(x, x_i) + b^*\right)$$

称为非线性支持向量机，$K(x,z)$是正定核函数。

下面叙述非线性支持向量机学习方法。

算法**13.7.5(**非线性支持向量机学习算法**)**

输入：训练数据集$T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$, 其中,

$x_i \in X = R^n, y_i \in Y = \{-1, +1\}, i = 1, 2, ..., N$;

输出：分类决策函数。

(1)选取适当的核函数$K(x,z)$和适当的参数C, 构造并求解最优化问题

$$\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K(x_i \cdot x_j) - \sum_{i=1}^{N} \alpha_i s.t. \quad \sum_{i=1}^{N} \alpha_i y_i = 00 \le \alpha_i \le C, i = 1, 2, \ \dots, N$$

求得最优解$\alpha^* = (\alpha_1^*, \alpha_2^*, \ \dots, \alpha_N^*)^T$

(2)选择α^*的一个正分量$0 < \alpha_j^* < C$, 计算

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

(3)求得分类决策函数：

$$f(x) = sign(\sum_{i=1}^{N} \alpha_i^* y_i K(x, x_i) + b^*)$$

当$K(x,z)$是正定核函数时，该问题存在解。

13.7.4 序列最小优化算法

前面我们讨论了各种问题的支持向量机的求解，其本质上都是求解对应凸二次规划问题的解。求解凸二次规划问题的方法有很多，但是当训练样本容量较大时，多数算法往往变得低效，这里介绍一种快速实现方法——序列最小最优化（SMO）。

算法**13.7.6(SMO**算法**)**

输入：训练数据集$T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$, 其中,

$x_i \in X = R^n, y_i \in Y = \{-1, +1\}, i = 1, 2, ..., N$, 精度$\varepsilon$;

输出：近似解$\hat{\alpha}$。

(1)取初值$\alpha^{(0)} = 0$, 令$k = 0$;

(2)选取优化变量$\alpha_1^{(k)}, \alpha_2^{(k)}$, 解析求解两个变量的最优化问题

$$W(\alpha_1, \alpha_2) = \frac{1}{2} K_{11}\alpha_1^2 + \frac{1}{2}K_{22}\alpha_2^2 + y_1 y_2 K_{12}\alpha_1\alpha_2 - \quad (\alpha_1 + \alpha_2) + y_1\alpha_1 \sum_{i=3}^{N} y_i\alpha_i K_{i1} + y_2\alpha_2 \sum_{i=3}^{N} y_i\alpha_i K_{i2} s.t. \quad \alpha_1 y_1 +$$

求得最优解$\alpha_1^{(k+1)}, \alpha_2^{(k+1)}$, 更新 α 为 $\alpha^{(k+1)}$;

(3)若在精度 ε 范围内满足停机条件

$$\sum_{i=1}^{N} \alpha_i y_i = 0,\ 0 \le \alpha_i \le C,\ i = 1, 2, ..., N y_i \cdot g(x_i) \{\ge 1, \quad \{x_i | \alpha_i = 0\} = 1, \quad \{0 < \alpha_i < C\} \le 1, \quad \{x_i | \alpha_i = C$$

其中,

$$g(x_i) = \sum_{j=1}^{N} \alpha_j y_j K(x_j, x_i) + b$$

则转（4）；否则令$k = k + 1$, 转（2）；

(4)取$\hat{\alpha} = \alpha^{(k+1)}$。

例题:编程实现SMO算法。

```
01 import numpy as np
02 def load_data(fname):
03     with open(fname, 'r') as f:
04         data = []
05         line = f.readline()
06         for line in f:
07             line = line.strip().split()
08             x1 = float(line[0])
09             x2 = float(line[1])
10             t = int(line[2])
11             data.append([x1, x2, t])
12         return np.array(data)
13
14 def eval_acc(label, pred):
15       return np.sum(label == pred) / len(pred)
16 # 定义线性核函数
17 class LinearKernel(object):
18     def __call__(self, x, y):
19         return np.dot(x, y.T)
20 # 定义SVM类实现SMO算法
21 class SVM(object):
22     def __init__(self,
23                  trainX,
24                  trainY,
```

```
25              C=1,
26              kernel=None,
27              differ=1e-3,
28              max_iter=100):
29          self.C = C  #正则化的参数
30          self.differ = differ  #用来判断是否收敛的阈值
31          self.max_iter = max_iter  #迭代次数的最大值
32          self.m = trainX.shape[0]
33          self.n = trainX.shape[1]
34          if kernel is None:
35              self.kernel = LinearKernel()  # 无核默认是线性的核
36          else:
37              self.kernel = kernel
38          self.weights = np.zeros(self.n)  # 权重
39          self.bias = 0  # 偏置值
40          self.alpha = np.zeros(self.m)  # 拉格朗日乘子
41          self.K = np.zeros((self.m, self.m))  # 特征经过核函数转化的值
42          self.X = trainX
43          self.Y = trainY
44          for i in range(self.m):
45              self.K[:, i] = self.kernel(self.X, self.X[i, :])
46  #每一行数据的特征通过核函数转化 n->m
47      def train(self):
48          for iter_ in range(self.max_iter):
49              alpha_prev = np.copy(self.alpha)
50              for j in range(self.m):
51                  #选择第二个优化的拉格朗日乘子
52                  i = self.random_index(j)
53                  error_i = self.error(i)
54                  error_j = self.error(j)
55                  #检验是否满足KKT条件，然后选择违反KKT条件最严重的alpha[j]
56                  if (self.Y[j] * error_j < -0.001 and
57                          self.alpha[j] < self.C) or (self.Y[j] * error_j > 0.001 and
58                          self.alpha[j] > 0):
59                      eta = 2.0 * self.K[i, j] - self.K[i, i] - self.K[j, j]
60                      #第j个要优化的拉格朗日乘子，最后需要的
61                      if eta >= 0:
62                          continue
63                      L, H = self.alpha_constrain(i, j)
64                      old_alpha_j, old_alpha_i = self.alpha[j], self.alpha[i]
65                      #旧的拉格朗日乘子的值
66                      self.alpha[j] -= (self.Y[j] * (error_i - error_j)) / eta  #self.alpha[j]更新
67
68                      #根据约束最后更新拉格朗日乘子self.alpha[j]，并且更新alpha[j]
69                      self.alpha[j] = self.alpha_new(self.alpha[j], H, L)
70                      self.alpha[i] = self.alpha[i] + self.Y[i] * self.Y[j] * (old_alpha_j -
71                                                  self.alpha[j])
72                      #更新偏置值b
73                      b1 = self.bias - error_i - self.Y[i] * (self.alpha[i] -
74                          old_alpha_j) * self.K[i, i] - self.Y[j] * (
```

```
75                      self.alpha[j] - old_alpha_j) * self.K[i, j]
76                  b2 = self.bias - error_j - self.Y[j] * (self.alpha[j] -
77                      old_alpha_j) * self.K[j, j] - self.Y[i] * (
78                          self.alpha[i] - old_alpha_i) * self.K[i, j]
79                  if 0 < self.alpha[i] < self.C:
80                      self.bias = b1
81                  elif 0 < self.alpha[j] < self.C:
82                      self.bias = b2
83                  else:
84                      self.bias = 0.5 * (b1 + b2)
85              #判断是否收敛
86              diff = np.linalg.norm(self.alpha - alpha_prev)
87              if diff < self.differ:
88                  for n in range(self.n):
89                      self.weights[n] = np.sum(self.alpha * self.Y * self.X[:, n])
90                  break
91
92      #随机一个要优化的拉格朗日乘子，该乘子必须和循环里面选择的乘子不同
93      def random_index(self, index):
94          index_array = np.array(range(self.m))
95          new_index = np.random.randint(len(np.delete(index_array, index)))
96          return new_index
97
98      def error(self, i):
99          k_v = self.kernel(self.X, self.X[i])
100         y_pred = np.dot((self.alpha * self.Y).T, k_v.T) + self.bias
101         return y_pred - self.Y[i]
102
106     def alpha_constrain(self, i, j):                                    #计算
alpha[j]范围约束
107         if self.Y[i] != self.Y[j]:                                      #当
y1≠y2
108             L = max(0, self.alpha[j] - self.alpha[i])
109             H = min(self.C, self.C - self.alpha[i] + self.alpha[j])
110         else:
            #当y1=y2
111             L = max(0, self.alpha[i] + self.alpha[j] - self.C)
112             H = min(self.C, self.alpha[i] + self.alpha[j])
113         return L, H
114
115
116     def alpha_new(self, alpha, H, L):                   #根据不等式约束条件
L≤alpha[j]_new≤H
117         if alpha > H:
118             alpha = H
119         elif alpha < L:
120             alpha = L
121         return alpha
122
123     def predict(self, x):
```

```
        pred = np.zeros(x.shape[0])
        for i in range(len(x)):
            dd = x[i, :]
            pred[i] = np.sign(np.dot(self.weights, dd) + self.bias)
        return pred
```

最后，读取数据，通过SMO算法实现SVM。

```
if __name__ == '__main__':
    # 载入数据，实际实用时将x替换为具体名称
    train_file = 'data/train_linear.txt'
    test_file = 'data/test_linear.txt'
    data_train = load_data(train_file)  # 数据格式[x1, x2, t]
    data_test = load_data(test_file)
    X_train = data_train[:, :2]
    y_train = data_train[:, 2]
    # 使用训练集训练SVM模型
    kernel = LinearKernel()
    svm = SVM(X_train, y_train, max_iter=500, kernel=kernel, C=0.6)  # 初始化模型
    svm.train()  # 训练模型

    # 使用SVM模型预测标签
    x_train = data_train[:, :2]  # feature [x1, x2]
    t_train = data_train[:, 2]  # 真实标签
    t_train_pred = svm.predict(x_train)  # 预测标签
    x_test = data_test[:, :2]
    t_test = data_test[:, 2]
    t_test_pred = svm.predict(x_test)

    # 评估结果，计算准确率
    acc_train = eval_acc(t_train, t_train_pred)
    acc_test = eval_acc(t_test, t_test_pred)
    print("train accuracy: {:.1f}%".format(acc_train * 100))
    print("test accuracy: {:.1f}%".format(acc_test * 100))
```

```
train accuracy: 95.0%
test accuracy: 94.5%
```

13.8 提升方法

提升方法是一种常用的统计学习方法。提升方法的思路为：对于一个复杂任务，将多个专家做出的判断结果进行适当的综合所得到的的结果，要比其中任意一个专家的判断好，即“三个臭皮匠顶个诸葛亮”的道理。下面我们将深入科学地对以上结论进行讨论介绍。

13.8.1 提升方法基本概念（AdaBoost）

历史上Kearns和Valiant提出了“强可学习”和“弱可学习”的概念。即为：在概率近似正确（PAC）学习的框架中，一个概念（一个类），如果存在一个多项式的学习算法能够学习它，并且有着很高的正确率，则称该概念是强可学习的；若算法正确率仅比随机猜测略好，则称该概念是弱可学习的。并且可以证得：强可学习和弱可学习是等价的，在PAC学习框架下，一个概念是强可学习

的充要条件是该概念是弱可学习的。这样我们可以通过将已经发现的“弱学习算法”提升（boost）为“强学习算法”。现如今已经发现了很多提升方法，在这些提升方法中最具代表性的是AdaBoost算法。

对于分类问题而言，提升方法从弱学习算法出发，学习到一系列弱分类器（基本分类器），然后组合这些弱分类器，构成一个强分类器。其中，学习弱分类器的过程通常是通过改变训练数据的概率分布（权值分布）实现的。

所以提升方法要解决两个问题：一、如何在每一轮中改变训练数据的概率分布；二、如何将弱分类器组合成一个强分类器。

AdaBoost的解决方案是：在每一轮的训练中，提高前一轮被弱分类器错误分类的样本点数据的权值，并且降低被弱分类器正确分类的样本点数据的权值，这样可以使得被错误分类的样本点在后一轮弱分类器的学习中获得更多的关注；弱分类器组合方面，采用加权多数表决的方法，加大分类误差率小的弱分类器的权值，使其在表决中起更大作用，并且减小分类误差率大的弱分类器的权值。

13.8.2 AdaBoost算法

算法13.8.1 (AdaBoost)

输入：训练数据集$\{(x_1, y_1), (x_2, y_2)\dots, (x_N, y_N)\}$，其中$x_i \in R^n$，$y_i \in \{-1, +1\}$，$i = 1, 2, .., N$；弱学习算法；

输出：最终分类器$G(x)$。

（1）初始化训练数据的权值分布

$$D_1 = \left(\omega_{11}, \dots, \omega_{1i}, \dots, \omega_{1N}\right), \quad \omega_{1i} = \frac{1}{N}, \quad i = 1, 2, \dots, N$$

（2）对$m = 1, 2, \dots, M$

（a）使用具有权值分布D_m的训练数据集学习，得到基本分类器

$$G_m(x): X \to \{-1, +1\}$$

（b）计算$G_m(x)$在训练数据集上的分类误差率

$$e_m = \sum_{i=1}^{N} P\left(G_m\left(x_i\right) \neq y_i\right) = \sum_{i=1}^{N} \omega_{mi} I\left(G_m\left(x_i\right) \neq y_i\right)$$

（c）计算$G_m(x)$的系数

$$\alpha_m = \frac{1}{2} log \frac{1 - e_m}{e_m}$$

这里的对数为自然对数。

（d）更新训练数据集的权值分布

$$D_{m+1} = \left(\omega_{m+1,1}, \dots, \omega_{m+1,i}, \dots, \omega_{m+1,N}\right)$$

$$\omega_{m+1,i} = \frac{\omega_{mi}}{Z_m} \exp exp\left(-\alpha_m y_i G_m\left(x_i\right)\right), \quad i = 1, 2, \dots, N$$

这里Z_m是规范化因子，即

$$Z_m = \sum_{i=1}^{N} \left(-\alpha_m y_i G_m\left(x_i\right)\right)$$

它使得D_{m+1}成为一个合法概率分布。

（3）构建基本分类器的线性组合

$$f(x) = \sum_{m=1}^{M} \alpha_m G_m(x)$$

得到最终分类器

$$G(x) = sign(f(x)) = sign\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$$

对AdaBoost算法作以下说明：

步骤（1）假设训练数据集具有均匀的权值分布，即每个训练样本在基本训练器中作用相同，这一假设保证第1步能够在原始数据上学习基本分类器$G_1(x)$。

步骤（2）AdaBoost反复学习基本分类器，在每一轮$m = 1, 2, ..., M$顺次地执行下列操作：

(a)使用当前分布D_m加权的训练数据集，学习基本分类器$G_m(x)$。

(b)计算基本分类器$G_m(x)$在加权训练数据集上的分类误差率：

$$e_m = \sum_{i=1}^{N} P\left(G_m\left(x_i\right) \neq y_i\right) = \sum_{G_m\left(x_i\right) \neq y_i} \omega_{mi}$$

这里ω_{mi}表示第m轮中第i个实例点的权值，$\sum_{i=1}^{N} \omega_{mi} = 1$。这表明$G_m(x)$在加权的训练数据集上的分类误差率是被$G_m(x)$误分类样本的权值之和，由此可看出数据权值分布$D_m$与基本分类器$G_m(x)$的分类误差率之间的关系。

(c)计算基本分类器$G_m(x)$的系数α_m。α_m表示$G_m(x)$在最终分类器中的重要性。

由$\alpha_m = \frac{1}{2} log \frac{1-e_m}{e_m}$可知，当$e_m \leq 0.5$时，$\alpha_m \geq 0$，并且$\alpha_m$随$e_m$的减小而增大，故而分类误差率越小的基本分类器在最终分类器中的作用越大。

(d)更新训练数据集权值分布为下一轮训练学习做准备，即有：

$$\omega_{m+1,i} = \{\frac{\omega_{mi}}{Z_m} e^{-\alpha_m},\ G_m\left(x_i\right) = y_i \quad \frac{\omega_{mi}}{Z_m} e^{\alpha_m},\ G_m\left(x_i\right) \neq y_i$$

所以有，被正确分类的样本点的权值减小，误分类样本点的权值得以增加，从而在下一轮的训练中得到更多的注意。

步骤（3）最终分类器$f(x)$通过对基本分类器进行线性组合，实现加权表决得出结果，其中α_m表示对应的基本分类器的重要性。注意到所有基本分类器的系数α_m之和并不一定等于1。$f(x)$的符号表明实例点的分类结果，其绝对值表明分类的确信度。

此外，可以证得AdaBoost算法的训练误差是以指数速率下降的，这也是AdaBoost算法的一个非常明显的优点。

例题：编程实现AdaBoost用于分类问题的形式，求最终分类器

首先，引入基本的库函数并获取数据。

```
01 import numpy as np
02
03 def loadDataSet():
04     dataSet = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]
05     label = [1, 1, 1, -1, -1, -1, 1, 1, 1, -1]
06     return np.mat(dataSet).T, np.mat(label)
```

构造具体实现过程。

```
01 def splitDataSet(dataMat, feat, value, comp, m):
02     # 满足 dataMat "comp" value 的位置值为1.0，不满足为-1.0
03     retArray = np.ones((m, 1))
04     if comp == ' larger than ':
05         retArray[dataMat[:, feat] < value] = -1.0
06     else:
07         retArray[dataMat[:, feat] > value] = -1.0
08     return retArray
09
```

```
10 def decisionTree(dataSet, labelList, weight):
11     dataMat = np.mat(dataSet)
12     labelMat = np.mat(labelList).T
13     bestFeat = {}
14     minError = np.inf
15     m, n = np.shape(dataMat)
16     bestClass = np.mat(np.zeros((m, 1)))
17     for i in range(n):
18         sortedIndex = np.argsort(dataMat, axis=i)
19         for j in range(m - 1):
20             value = (dataMat[sortedIndex[j], i] +
21                      dataMat[sortedIndex[j + 1], i]) / 2.0
22             for comp in ['larger than', 'smaller than']:  # 符号可以是大于或者小于
23                 retArray = splitDataSet(dataMat, i, value, comp, m)
24                 errSet = np.mat(np.ones((m, 1)))
25                 errSet[retArray == labelMat] = 0
26                 weightError = weight.T * errSet
27                 if weightError < minError:
28                     minError = weightError
29                     bestFeat['feat'] = i
30                     bestFeat['split_value'] = float(value)
31                     bestFeat['comp'] = comp
32                     bestClass = retArray.copy()
33     return bestFeat, minError, bestClass
34
35 def adaBoostTrain(dataSet, label, numIt=10):
36     classifDict = []
37     m = np.shape(dataSet)[0]
38     totalRetResult = np.mat(np.zeros((m, 1)))
39     weight = np.mat(np.ones((m, 1)) / m)
40     for _ in range(numIt):
41         bestFeat, error, EstClass = decisionTree(dataSet, label, weight)
42         alpha = float(0.5 * np.log((1 - error) / error))  # 该弱分类器的系数
43         bestFeat['alpha'] = alpha
44         classifDict.append(bestFeat)
45         wtx = np.multiply(-1 * alpha * np.mat(label).T, EstClass)
46         weight = np.multiply(weight, np.exp(wtx))
47         weight = weight / sum(weight)
48         totalRetResult += alpha * EstClass
49         totalError = (sum(label.T != np.sign(totalRetResult))) / float(m)
50         if totalError == 0: break
51     return classifDict, totalRetResult  # 返回最终分类器
```

最后，调用以上函数完成基于AdaBoost算法的分类。

```
01 dataSet, label = loadDataSet()
02 classifDict, totalRetResult = adaBoostTrain(dataSet, label)
03 print("classifDict", classifDict)
04 print(np.sign(totalRetResult))
```

```
01 classifDict [{'feat': 0, 'split_value': 2.5, 'comp': 'smaller than', 'alpha': 0.4236489301936017},
```

```
02 {'feat': 0, 'split_value': 8.5, 'comp': 'smaller than', 'alpha': 0.6496414920651304}, {'feat': 0,
03 'split_value': 5.5, 'comp': 'larger than', 'alpha': 0.752038698388137}]
04 [[ 1.]
05  [ 1.]
06  [ 1.]
07  [-1.]
08  [-1.]
09  [-1.]
10  [ 1.]
11  [ 1.]
12  [ 1.]
13  [-1.]]
```

根据结果可得，所学习到的最终分类器为

$$G(x) = sign\left(0.4236G_1(x) + 0.6496G_2(x) + 0.7520G_3(x)\right)$$

13.8.3 AdaBoost算法的解释

AdaBoost算法还有另一个解释，即可认为AdaBoost算法是模型为加法模型、损失函数为指数损失函数、学习算法为前向分步算法时的二类分类学习方法。

下面介绍前向分步算法，注意和上一节AdaBoost算法进行对比学习。

算法13.8.2 (前向分步算法)

输入：训练数据集$\{(x_1, y_1), (x_2, y_2)\ldots, (x_N, y_N)\}$，损失函数$L(y, f(x))$，基函数集$\{b(x;\gamma)\}$；

输出：加法模型$f(x)$。

(1)初始化$f_0(x) = 0$；

(2)对$m = 1, 2, \ldots, M$

(a)极小化损失函数

$$\left(\beta_m, \gamma_m\right) = arg \sum_{i=1}^{N} L\left(y_i, f_{m-1}\left(x_i\right) + \beta b\left(x_i;\gamma\right)\right)$$

得到参数β_m, γ_m。

(b)更新

$$f_m(x) = f_{m-1}(x) + \beta_m b(x;\gamma)$$

(3)得到加法模型

$$f(x) = f_M(x) = \sum_{m=1}^{M} \beta_m b\left(x;\gamma_m\right)$$

这样，前向分步算法将同时求解从$m = 1$到M所有参数β_m, γ_m的优化问题简化为逐次求解各个β_m, γ_m的优化问题。

13.8.4 提升树

提升树是以分类树或回归树作为基本分类器的提升方法，被认为是统计学习中性能最好的方法之一。

提升树模型可表示为决策树的加法模型：

$$f_M(x) = \sum_{m=1}^{M} T(x;\Theta_m)$$

其中，$T(x;\Theta_m)$表示决策树，Θ_m表示决策树的参数，M为树的个数。

算法13.8.3 (回归问题的提升树算法)

输入：训练数据集$\{(x_1, y_1), (x_2, y_2)\ldots, (x_N, y_N)\}$，其中$x_i \in X \subseteq \boldsymbol{R}^n$，$y_i \in Y \subseteq \boldsymbol{R}$，$i$=1, 2, .., N；

输出：提升树$f_M(x)$。

(1)初始化$f_0(x) = 0$;

(2)对$m = 1, 2, ..., M$

(a)计算残差：

$$r_{mi} = y_i - f_{m-1}(x_i), \ i = 1, 2, ..., N$$

(b)拟合残差r_{mi}学习一个回归树，得到$T(x; \Theta_m)$

(c)更新

$$f_m(x) = f_{m-1}(x) + T(x; \Theta_m)$$

(3)得到回归问题的提升树

$$f_M(x) = \sum_{m=1}^{M} T(x; \Theta_m)$$

例题:编程实现用于解决回归问题的提升树模型

用封装一个BoostingTree类用于实现提升树模型

```
01 import numpy as np
02
03 class BoostingTree(object):
04     def __init__(self, train_samples, max_err):
05         self.Samples = train_samples
06         self.max_e = max_err
07         self.rsd = []  #shape=(N, ) 残差表
08         self.S = []  #shape=(1,N-1) 训练样本的切分点集合
09         self.FX = []  #shape(M, 3)  e=[s,c1,c2]表示一个回归树
10         self.init_S()
11         self.build_boostingTree()
12
13     def init_S(self):
14         '''计算所有可能切分点的集合，self.S'''
15         X, N = self.Samples[:, 0], np.shape(self.Samples)[0]
16         for i in np.arange(N - 1):
17             self.S.append((X[i] + X[i + 1]) / 2.0)
18         return self.S
19
20     def build_boostingTree(self):
21         if 0 == np.shape(self.FX)[0]:
22             self.rsd = self.Samples[:, 1]  # 初始残差表为标签值列表
23         X, Y = self.Samples[:, 0], self.rsd
24         C = []  # shape=(N-1,2) 记录各切分点情况下的R1和R2的平均回归值
25         M = []  #len=N-1, ms=e=[m1,m2,...,mN] 记录各切分点情况下的回归误差平方和
26         for i in range(np.shape(self.S)[0]):
27             #记录R1和R2区域的平均回归值
28             RY1, RY2 = Y[0:i + 1], Y[i + 1:]
29             C.append([np.mean(RY1), np.mean(RY2)])
30             #求切分点的回归误差平方和m
31             m1_ufunc = lambda y: np.power(y - C[i][0], 2)
32             m2_ufunc = lambda y: np.power(y - C[i][1], 2)
33             m1_func = np.frompyfunc(m1_ufunc, 1, 1)
```

```
            m2_func = np.frompyfunc(m2_ufunc, 1, 1)
            cur_m = m1_func(RY1).sum() + m2_func(RY2).sum()
            M.append(cur_m)
        #生成回归树
        i = np.argmin(M)
        s, c1, c2 = self.S[i], C[i][0], C[i][1]  # 记录回归树的切分点和两侧区域的回归值
        #更新提升树
        self.FX.append([s, c1, c2])
        #更新残差表(用提升树拟合训练数据的残差表)
        rsd_ufunc = lambda x, y: (x < s and [y - c1] or [y - c2])[0]
        rsd_func = np.frompyfunc(rsd_ufunc, 2, 1)
        self.rsd = rsd_func(X, Y)
        #计算用提升树拟合训练数据的平方损失误差
        e_ufunc = lambda r: np.power(r, 2)  #r为残差
        e_func = np.frompyfunc(e_ufunc, 1, 1)
        e = e_func(self.rsd).sum()
        #判断是否终止生成提升树
        if e > self.max_e:  #大于设置的误差阈值，则继续生成提升树
            return self.build_boostingTree()
        else:
            print('BoostingTree succeed as:\n', np.array(self.FX))
            return self.FX

    def Regress(self, test_samples):
        base_tree = lambda x, v, f1, f2: (x < v and [f1] or [f2])[0]
        reg_efunc = lambda x: sum([base_tree(x, v, f1, f2) for v, f1, f2 in self.FX])
        reg_func = np.frompyfunc(reg_efunc, 1, 1)
        regress_result = reg_func(test_samples)
        return regress_result
```

通过实例化BoostingTree类对象学习提升树模型，得到预测结果

```
train_samples = np.array([[1, 5.56], [2, 5.70], [3, 5.91], [4, 6.40], [5, 6.80],
                  [6, 7.05], [7, 8.90], [8, 8.70], [9, 9.00], [10, 9.05]])
bt = BoostingTree(train_samples, max_err=0.2)
test_samples = train_samples[:, 0]
ret = bt.Regress(test_samples)
print('boostingTree regress result:\n', ret)
```

```
BoostingTree succeed as:
 [[ 6.5         6.23666667  8.9125    ]
 [ 3.5        -0.51333333   0.22      ]
 [ 6.5         0.14666667  -0.22      ]
 [ 4.5        -0.16083333   0.10722222]
 [ 6.5         0.07148148  -0.10722222]
 [ 2.5        -0.15064815   0.03766204]]
boostingTree regress result:
 [5.63 5.63 5.818310185185186 6.551643518518518 6.819699074074074
 6.819699074074074 8.950162037037037 8.950162037037037 8.950162037037037
 8.950162037037037]
```

13.8.5 梯度提升

提升树是一个机器学习中较为高效可靠的方法，但是当损失函数较为复杂时，优化方面较为困难。可以将损失函数的负梯度在当前模型的值作为提升树算法中残差的近似值，这样拟合回归树的过程会容易许多。

算法13.8.4 (梯度提升算法)

输入：训练数据集$\{(x_1, y_1), (x_2, y_2)\ldots, (x_N, y_N)\}$，其中$x_i \in X \subseteq \boldsymbol{R}^n$，$y_i \in Y \subseteq \boldsymbol{R}$，$i=1, 2, .., N$，损失函数$L(y, f(x))$；

输出：回归树$\hat{f}(x)$。

(1)初始化

$$f_0(x) = arg \sum_{i=1}^{N} L(y_i, c)$$

(2)对$m = 1, 2, ..., M$

(a)对$i = 1, 2, ..., N$，计算残差：

$$r_{mi} = -\left(\frac{\partial L\left(y_i, f\left(x_i\right)\right)}{\partial f\left(x_i\right)}\right)_{f(x)=f_{m-1}(x)}$$

(b)对r_{mi}拟合一个回归树，得到第m棵树的叶节点区域R_{mj}，$j = 1, 2, ..., J$。

(c)对$j = 1, 2, ..., J$，计算

$$c_{mj} = arg \sum_{x_i \in R_{mj}} L\left(y_i, f_{m-1}\left(x_i\right) + c\right)$$

(d)更新$f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J} c_{mj} I(x \in R_{mj})$

(3)得到回归树

$$\hat{f}(x) = f_M(x) = \sum_{m=1}^{M} \sum_{j=1}^{J} c_{mj} I(x \in R_{mj})$$

算法第1步初始化，估计使损失函数极小化的常数值，它是只有一个根节点的树。第2（a）步计算损失函数的负梯度在当前模型的值，将它作为残差的估计。对于平方损失函数，它就是通常所说的残差；对于一般损失函数，它就是残差的近似值。第2（b）步估计回归树叶节点区域，以拟合残差的近似值。第2（c）步利用线性搜索估计叶节点区域的值，使损失函数极小化。第2（d）步更新回归树。第3步得到输出的最终模型$\hat{f}(x)$。

13.9 EM算法及其推广

EM算法是一种迭代算法，用于含有隐变量的概率模型参数的极大似然估计，或极大后验概率估计。EM算法主要由两部分组成：E步（expectation），求期望；M步（maximization），求极大。下面将对EM算法及其推广进行较为详细的讲解。

13.9.1 EM算法

概率模型有时既含有观测变量，又含有隐变量或潜在变量。如果概率模型的变量都是观测变量，那么给定数据，可以直接用极大似然估计法，或贝叶斯估计法估计模型参数。但是，当模型含有隐变量时，就不能简单地使用这些估计方法。EM算法就是含有隐变量的概率模型参数的极大似然估计法，或极大后验概率估计法。我们仅讨论极大似然估计，极大后验概率估计与其类似。

一般地，用Y表示观测随机变量的数据，Z表示隐随机变量的数据。Y和Z连在一起称为完全数据，观测数据Y又称为不完全数据。假设给定观测数据Y，其概率分布是$P(Y|\theta)$，其中θ是需要估计的模型参数，那么不完全数据Y的似然函数是$P(Y|\theta)$，对数似然函数$L(\theta) = log\, P(Y|\theta)$；假设Y和Z的联合概率分布是$P(Y,Z|\theta)$，那么完全数据的对数似然函数是$log\, P(Y,Z|\theta)$。

EM算法通过迭代求$L(\theta) = log\, P(Y|\theta)$的极大似然估计。每次迭代包含两步：E步，求期望M步，求极大化。下面先引入Q函数的概念，再详细叙述EM算法的过程。

定义**13.9.1 (Q函数)**

完全数据的对数似然函数$log\, P(Y,Z|\theta)$关于在给定观测数据Y和当前参数$\theta^{(i)}$下对未观测数据Z的条件概率分布$P(Z|Y,\theta^{(i)})$的期望称为Q函数，即

$$Q\left(\theta,\theta^{(i)}\right) = E_Z\left(\log log\ P(Y,Z|\theta)|Y,\ \theta^{(i)}\right)$$

算法13.9.1 (EM算法)

输入：观测变量数据Y，隐变量数据Z，联合分布$P(Y,Z|\theta)$，条件分布$P(Z|Y,\theta)$；

输出：模型参数θ。

(1)选择参数的初值$\theta,\theta^{(0)}$，开始迭代：

(2)E步：记$\theta^{(i)}$为第i次迭代参数θ的估计值，在第i + 1次迭代的E步，计算

$$Q\left(\theta,\theta^{(i)}\right) = E_Z\left(\log log\ P(Y,Z|\theta)|Y,\ \theta^{(i)}\right) = \sum_Z \log log\ P(\theta)\, P(Z|Y,\theta^{(i)})$$

这里，$P(Z|Y,\theta^{(i)})$是在给定观测数据Y和当前参数估计$\theta^{(i)}$下隐变量数据Z的条件概率分布；

(3)M步：求使得$Q\left(\theta,\theta^{(i)}\right)$极大化的$\theta$，确定第i+1次迭代的参数估计值$\theta^{(i+1)}$

$$\theta^{(i+1)} = argQ\left(\theta,\theta^{(i)}\right)$$

(4)重复第(2)步和第(3)步，直到收敛。

下面关于EM算法作几点说明：

步骤（1）参数的初值可以任意选择，但需注意EM算法对初值是敏感的。

步骤（2）E步求$Q\left(\theta,\theta^{(i)}\right)$。Q函数式中Z是未观测数据，Y是观测数据。注意$Q\left(\theta,\theta^{(i)}\right)$的第1个变元表示要极大化的参数，第2个变元表示参数的当前估计值。每次迭代实际在求Q函数及其极大。

步骤（3）M步求$Q\left(\theta,\theta^{(i)}\right)$的极大化，得到$\theta^{(i+1)}$，完成一次迭代$\theta^{(i)} \to \theta^{(i+1)}$。可以证得每次迭代使似然函数增大或达到局部极值。

步骤（4）给出停止迭代的条件，一般是对较小的正数$\varepsilon_1,\varepsilon_2$，若满足

$$\|\theta^{(i+1)} - \theta^{(i)}\| < \varepsilon_1 \text{ 或 } \|Q\left(\theta^{(i+1)},\theta^{(i)}\right) - Q\left(\theta^{(i)},\theta^{(i)}\right)\| < \varepsilon_2$$

则停止迭代。

例题（三硬币模型假设有3枚硬币，分别记作A，B，C。这些硬币正面出现的概率分别是p和q。进行如下掷硬币试验：先掷硬币A，根据其结果选出硬币B或硬币C，正面选硬币B，反面选硬币C；然后掷选出的硬币，掷硬币的结果，出现正面记作1，出现反面记作0；独立地重复n次试验(这里，n = 10)，观测结果如下：

1,1,0,1,0,0,1,0,1,1

假设只能观测到掷硬币的结果，不能观测掷硬币的过程。问如何估计三硬币正面出现的概率，即三硬币模型的参数。

首先引入库函数和观测数据。

```
01 import numpy as np
```

```
02
03 def load_data():
04     Y = np.array([1, 1, 0, 1, 0, 0, 1, 0, 1, 1])
05     return Y
```

通过定义EM类实现对该问题参数的求解。

```
01 class EM():
02     def __init__(self, data, params=(0.5, 0.5, 0.5), max_err=0.01):
03         self.data = data
04         self.max_err = max_err
05         self.params = params
06         self.method(self.params)
07
08     def method(self, params):
09         Y = self.data
10         π, p, q = params
11         p_B = (π * np.power(p, Y) * np.power(1 - p, 1 - Y))
12         p_C = ((1 - π) * np.power(q, Y) * np.power(1 - q, 1 - Y))
13         μ = p_B / (p_B + p_C)
14         # 更新参数
15         π_new = μ.sum() / Y.shape[0]
16         p_new = (μ * Y).sum() / μ.sum()
17         q_new = ((1 - μ) * Y).sum() / (1 - μ).sum()
18         loss = (π_new - π) + (p_new - p) + (q_new - q)
19         if loss < self.max_err:
20             print(f'Final params is [{π_new, p_new, q_new}]')
21             return
22         else:
23             self.method(params=(π_new, p_new, q_new))
24
25 estimator = EM(load_data())
```

```
01 Final params is [(0.5, 0.6, 0.6)]
```

下面尝试更改参数初值。

```
01 estimator = EM(load_data(), params=(0.4, 0.6, 0.7))
```

```
01 Final params is [(0.40641711229946526, 0.5368421052631579, 0.6432432432432431)]
```

由此可以直观得出，EM算法对于参数初值的选取很敏感。

13.9.2 EM算法在高斯混合模型中的应用

高斯混合模型应用广泛，是一个典型的含有隐变量的概率模型。EM可以很好地求解高斯混合模型。下面对高斯混合模型及其对应的EM算法进行介绍。

定义13.9.2 (高斯混合模型)

高斯混合模型是指具有如下形式的概率分布模型：

$$P(\theta) = \sum_{k=1}^{K} \alpha_k \varphi\left(\theta_k\right)$$

其中，α_k是系数，$\alpha_k \geq 0$，$\sum_{k=1}^{K} \alpha_k = 1$；$\varphi\left(\theta_k\right)$是高斯分布密度，$\theta_k = (\mu_k, \sigma_k^2)$，

$$\varphi\left(\theta_k\right)=\frac{1}{\sqrt{2\pi}\sigma_k}\exp\, exp\left(-\frac{\left(y-\mu_k\right)^2}{2\sigma_k^2}\right)$$

称为第k个分模型。

此外，将高斯混合模型中的高斯分布替换为任意概率分布可以得到更加一般化的混合模型。

算法13.9.2 (高斯混合模型参数估计的EM算法)

输入：观测数据$y_1, y_2, ..., y_N$，高斯混合模型；

输出：高斯混合模型参数。

(1)取参数的初始值开始迭代；

(2)E步：依据当前模型参数，计算分模型k对观测数据y_j的响应度

$$\hat{\gamma}_{jk}=\frac{\alpha_k\varphi\left(\theta_k\right)}{\sum_{k=1}^{K}\alpha_k\varphi\left(\theta_k\right)},\quad j=1,2,...,N;\quad k=1,2,...,K$$

(3)M步：计算新一轮迭代的模型参数

$$\hat{\mu}_k=\frac{\sum_{j=1}^{N}\hat{\gamma}_{jk}y_j}{\sum_{j=1}^{N}\hat{\gamma}_{jk}},\quad k=1,2,...,K$$

$$\hat{\sigma}_k^2=\frac{\sum_{j=1}^{N}\hat{\gamma}_{jk}\left(y_j-\mu_k\right)^2}{\sum_{j=1}^{N}\hat{\gamma}_{jk}},\quad k=1,2,...,K$$

$$\hat{\alpha}_k=\frac{\sum_{j=1}^{N}\hat{\gamma}_{jk}}{N},\quad k=1,2,...,K$$

(4)重复第(2)步和第(3)步，直到收敛。

13.9.3 EM算法的推广

EM算法还可以解释为F函数的极大极大算法，进而推广为广义期望极大算法（GEM）。下面对F函数的定义以及GEM算法的内容进行讲解。

定义13.9.3 (F函数)

假设隐变量数据Z的概率分布为$\tilde{P}(Z)$，定义分布$\tilde{P}$与参数θ的函数$F(\tilde{P},\theta)$如下：

$$F\left(\tilde{P},\theta\right)=E_{\tilde{P}}(\log log\, P(\theta))+H(\tilde{P})$$

称为F函数。式中$H\left(\tilde{P}\right)=-E_{\tilde{P}}\log log\,\tilde{P}(Z)$是分布$\tilde{P}(Z)$的熵。

算法13.9.3 (GEM算法1)

输入：观测数据，F函数；

输出：模型参数。

(1)初始化参数$\theta^{(0)}$，开始迭代；

(2)第i+1次迭代，第1步：记$\theta^{(i)}$为参数θ的估计值，$\tilde{P}^{(i)}$为函数$\tilde{P}$的估计，求$\tilde{P}^{(i+1)}$使$\tilde{P}$极大化$F\left(\tilde{P},\theta^{(i)}\right)$；

(3)第2步：求$\theta^{(i+1)}$使$F\left(\tilde{P}^{(i+1)},\theta\right)$极大化；

(4)重复(2)和(3)，直到收敛。

在GEM算法1中，有时求$Q\left(\theta,\theta^{(i)}\right)$的极大化是很困难的，可以在每一次迭代中找到$\theta^{(i+1)}$使得

$Q\left(\theta^{(i+1)}, \theta^{(i)}\right) > Q\left(\theta^{(i)}, \theta^{(i)}\right)$成立即可。

算法13.9.4 (GEM算法2)
输入：观测数据，Q函数；
输出：模型参数。
(1)初始化参数$\theta^{(0)}$，开始迭代；
(2)第i+1次迭代，第1步：记$\theta^{(i)}$为参数θ的估计值，计算

$$Q\left(\theta, \theta^{(i)}\right) = E_Z\left(\log log\ P(Y, Z|\theta)|Y,\ \theta^{(i)}\right) = \sum_Z \log log\ P(\theta)\ P(Z|Y, \theta^{(i)})$$

(3)第2步：求$\theta^{(i+1)}$使

$$Q\left(\theta^{(i+1)}, \theta^{(i)}\right) > Q\left(\theta^{(i)}, \theta^{(i)}\right)$$

(4)重复(2)和(3)，直到收敛。

在GEM算法中，有时参数θ的维度d较高 ($d \geq 2$)，较难计算。这时可以将EM算法的M步分解为d次条件极大化，每次只改变参数向量的一个分量，其余分量不改变。

算法13.9.5 (GEM算法3)
输入：观测数据，Q函数；
输出：模型参数。
(1)初始化参数$\theta^{(0)} = (\theta_1^{(0)}, \theta_2^{(0)}, \ldots, \theta_d^{(0)})$，开始迭代；
(2)第i+1次迭代，第1步：记$\theta^{(i)} = (\theta_1^{(i)}, \theta_2^{(i)}, \ldots, \theta_d^{(i)})$为参数$\theta = (\theta_1, \theta_2, \ldots, \theta_d)$的估计值，计算

$$Q\left(\theta, \theta^{(i)}\right) = E_Z\left(\log log\ P(Y, Z|\theta)|Y,\ \theta^{(i)}\right) = \sum_Z \log log\ P(\theta)\ P(Z|Y, \theta^{(i)})$$

(3)第2步：进行d次条件极大化：
首先，在$\theta_2^{(i)}, \ldots, \theta_d^{(i)}$保持不变的条件下求使得$Q\left(\theta, \theta^{(i)}\right)$达到极大的$\theta_1^{(i+1)}$；
然后，在$\theta_1 = \theta_1^{(i+1)}$，$\theta_j = \theta_j^{(i)}$，$j = 3, 4, \ldots, d$的条件下求使得$Q\left(\theta, \theta^{(i)}\right)$达到极大的$\theta_2^{(i+1)}$；
如此继续，经过d次条件极大化，得到$\theta^{(i+1)} = (\theta_1^{(i+1)}, \theta_2^{(i+1)}, \ldots, \theta_d^{(i+1)})$使得

$$Q\left(\theta^{(i+1)}, \theta^{(i)}\right) > Q\left(\theta^{(i)}, \theta^{(i)}\right)$$

(4)重复(2)和(3)，直到收敛。