感知机

1 感知机

1.1 线性可分

定义：给定数据集

$$T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$$

其中，$x_i \in R^n, y_i \in \{-1, 1\}, i = 1, 2, ..., N$，如果存在某个超平面 $S: \omega x + b = 0$，使得对 $\forall i = 1, 2, ..., N, (\omega x_i + b)y_i > 0$，则称数据集 T 为线性可分数据集。

感知机需要数据线性可分。

1.2 损失函数

感知机使用**误分类点**到超平面总距离作为学习的损失函数。记 M 是误分类点的集合，则感知机的损失函数为：

$$L(\omega, b) = -\frac{1}{||\omega||} \sum_{x_i \in M} y_i(\omega x_i + b)$$

不妨令 $||\omega|| = 1$(这是因为 $\omega x + b = 0$ 和 $\frac{\omega}{||\omega||}x + \frac{b}{||\omega||} = 0$ 是一个平面)，则感知机的损失函数为

$$L(\omega, b) = -\sum_{x_i \in M} y_i(\omega x_i + b)$$

1.3 优化算法

优化算法的目标是求得 $\omega^*, b^* = argmin_{\omega,b} L(\omega, b)$，使用随机梯度下降法进行优化。方法是：每次选出一个误分类点 (x_i, y_i)，对于该误分类点的损失函数 $L_i(\omega, b) = -y_i(\omega x_i + b)$，有：

$$\begin{cases} \frac{\partial L_i}{\partial \omega} = -y_i x_i \\ \frac{\partial L_i}{\partial b} = -y_i \end{cases}$$

因此，参数的更新过程为：

$$\begin{cases} \omega := \omega + \eta y_i x_i \\ b := b + \eta y_i \end{cases}$$

所以感知机算法如下：

输入：线性可分的训练数据集 $T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$。其中，$x_i \in R^n, y \in \{-1, 1\}, i = 1, 2, ..., N$ 和学习率 $\eta \in (0, 1]$。

1. 随机选取初值 w_0, b_0
2. 在训练集中选取数据 (x_i, y_i)
3. 若 $y_i(\omega x_i + b) \leq 0$,

$$\omega := \omega + \eta y_i x_i$$

$$b := b + \eta y_i$$

4. 回到 2，直至训练集中没有误分类点

输出：参数 ω, b 和对应的感知机模型 $f(x) = sign(\omega x + b)$。

1.4 收敛性

引理：对于将数据集 T 完全分开的超平面 $\omega_{opt} x + b_{opt} = 0$，不妨令其满足条件 $||\omega_{opt}|| = 1$。则存在 $\gamma > 0$，对所有 $i = 1, 2, ..., N$ 有：

$$y_i(\hat{\omega}_{opt} \hat{x}_i) := y_i(\omega_{opt} x + b_{opt}) \geq \gamma$$

Proof：显然。

定理：感知机算法必收敛。且其收敛次数 $k \leq (\frac{R}{\gamma})^2$，其中 $R = \max_{1 \leq i \leq N} ||\hat{x}_i||$。

Proof：

设 $\hat{\omega}_k$ 是误分点 (x_i, y_i) 在 $\hat{\omega}_{k-1}$ 上更新而来的。因此有 $y_i \hat{\omega}_{k-1} \hat{x}_i < 0$。

$$\begin{aligned} \hat{\omega}_k \hat{\omega}_{opt} &= (\hat{\omega}_{k-1} + \eta y_i x_i) \hat{\omega}_{opt} \\ &= \hat{\omega}_{k-1} \hat{\omega}_{opt} + \eta y_i x_i \hat{\omega}_{opt} \\ &\geq \hat{\omega}_{k-1} \hat{\omega}_{opt} + \eta \gamma \end{aligned}$$

假设 $\hat{\omega}_0 = 0$, 将此式不断递推即有：

$$\hat{\omega}_k\hat{\omega}_{opt} \geq \hat{\omega}_{k-1}\hat{\omega}_{opt} + \eta\gamma \geq \hat{\omega}_{k-2}\hat{\omega}_{opt} + 2\eta\gamma \geq ... \geq k\eta\gamma$$

此外，

$$\begin{aligned}
||\hat{\omega}_k||^2 &= ||\hat{\omega}_{k-1} + \eta y_i x_i||^2 \\
&= ||\omega_{k-1}||^2 + 2\eta y_i \hat{\omega}_{k-1}\hat{x}_i + \eta^2 x_i^2 \\
&\leq ||\omega_{k-1}||^2 + 0 + \eta^2 R^2 \\
&= ||\omega_{k-1}||^2 + \eta^2 R^2 \\
&\leq ||\omega_{k-2}||^2 + 2\eta^2 R^2 \leq ... \\
&\leq k\eta^2 R^2
\end{aligned}$$

联立以上两式有：

$$k\eta\gamma \leq \hat{\omega}_k\hat{\omega}_{opt} \leq ||\hat{\omega}_k|| \cdot ||\hat{\omega}_{opt}|| = ||\hat{\omega}_k|| \leq \sqrt{k}\eta R$$

由 $k\eta\gamma \leq \sqrt{k}\eta R$ 得 $k \leq (\frac{R}{\gamma})^2$。

1.5 对偶形式

原问题的参数更新形式是：

$$\omega := \omega + \eta y_i x_i$$

$$b := b + \eta y_i$$

因此，参数 ω 和 b 可以表示为如下线性组合：

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

$$b = \sum_{i=1}^{N} \alpha_i y_i$$

对照原问题的优化步骤，对偶问题的优化步骤为：

输入：线性可分的训练数据集 $T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$。其中，$x_i \in R^n, y \in \{-1, 1\}, i = 1, 2, ..., N$ **和**学习率 $\eta \in (0, 1]$。

1. 随机选取初值 $\alpha = (\alpha_1, \alpha_2, ..., \alpha_N)^T$ 和 b
2. 在训练集中选取数据 (x_i, y_i)
3. 若 $y_i[(\sum_{j=1}^{N} \alpha_j y_j x_j)x_i + b] \leq 0$,

$$\alpha_i := \alpha_i + \eta$$

$$b := b + \eta y_i$$

4. 回到 2，直至训练集中没有误分类点

输出：参数 α, b 和对应的模型 $f(x) = sign(\sum_{j=1}^{N} \alpha_j y_j x_j \cdot x + b)$。

在使用对偶问题求解时，由于不断使用 x_i 和 x_j 的积，所以常常使用 Gram 矩阵进行存储内积，即 $G = [x_i \cdot x_j]_{N \times N}$。

2 代码实现

考虑以下数据:

x	y
$(3,3)^T$	1
$(4,3)^T$	1
$(1,1)^T$	-1

2.1 sklearn 实现

准备数据：

```
import numpy as np
X = np.array([[3,3],[4,3],[1,1]])
Y = np.array([1,1,-1]).reshape(-1,1)
```

感知机的 sklearn 接口位于 sklearn.linear_model.Perceptron, 其文档可见**此处**。其中较为重要的参数有：

- eta0：感知机的学习率

```
from sklearn.linear_model import Perceptron
clf = Perceptron(eta0=1)
clf.fit(X,Y)
```

```
print(clf.coef_,clf.intercept_,clf.n_iter_)
```

```
## [[1. 0.]] [-2.] 9
```

2.2 原问题实现

```
def Perceptron(X,Y,eta=1):
    N,dim = X.shape
```

```
    w,b = np.zeros(dim),0
    it = 0
    point = 0
    while it<N:
        if (np.sum(X[point,:]*w)+b)*Y[point,:]<=0:
            w = w+eta*Y[point,:]*X[point,:]
            b = b+eta*Y[point,:]
            it = 0
        else:
            it += 1
        point = (point+1)%N
    return w,b
Perceptron(X,Y,eta=1)
```

```
## (array([1., 1.]), array([-3]))
```

2.3 对偶问题实现

```
def Perceptron_dual(X,Y,eta=1):
    N,_ = X.shape
    alpha,b = np.zeros(N),0
    G = X.dot(X.T)
    it = 0
    point = 0
    while it<N:
        if (np.sum(alpha*G[point,:]*Y.reshape(-1))+b)*Y[point,:]<=0:
            alpha[point] = alpha[point]+eta
            b = b+eta*Y[point,:]
            it = 0
        else:
            it += 1
        point = (point+1)%N
    return alpha,b
Perceptron_dual(X,Y,eta=1)
```

```
## (array([2., 0., 5.]), array([-3]))
```

k 邻近法

1 k 邻近法

1.1 距离度量

对于 $x_i, x_j \in R^n, x_i = (x_i^1, x_i^2, ..., x_i^n), x_i = (x_j^1, x_j^2, ..., x_j^n)$，两者之间的 L_p 距离定义为：

$$L_P(x_i, x_j) = (\sum_{l=1}^{n} |x_i^l - x_j^l|^p)^{\frac{1}{p}}$$

几种特殊的 L_p 距离：

- $p = 2$：欧氏距离

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

- $p = 1$：曼哈顿距离

$$L_1(x_i, x_j) = \sum_{l=1}^{n} |x_i^l - x_j^l|$$

- $p = +\infty$：各个坐标距离最大值

$$L_\infty(x_i, x_j) = \max |x_i^l - x_j^l|$$

1.2 k 邻近算法

k 邻近算法步骤如下：

输入：训练集 $T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$。其中，$x_i \in R^n, y \in \{c_1, c_2, ...c_K\}, i = 1, 2, ..., N$ 和实例特征 x。

1. 根据给定的距离度量方式 $l(x_i, x_j)$，在训练集 T 中找出与 x 最邻近的 k 个点，记这 k 个点所构成的集合为 $N_k(x)$
2. 决定 x 的类别

$$y = argmax_{c_j} \sum_{x_i \in N_k(x)} I(y_i = c_j), i = 1, 2.., N; j = 1, 2, ..., K$$

输出：x 的类别 y。

1.3 k 的选取

通常采用交叉验证法来选取最优的 k 值。

1.4 kd 树

kd 树可以大幅度地将 k 邻近法的**平均效率**从 $O(n)$ 提升到 $O(\log n)$。

1.4.1 构造 kd 树

kd 树不断地使用垂直于坐标轴的超平面将 k 维空间切分，将空间划分为一系列 k 维的超矩形区域。构造 kd 树的步骤如下：

1. 构造根节点，根节点对应包含所有实例点的超矩形区域
2. 对 k 维空间进行划分，生成子节点。划分方法是：(1) 在超矩形区域上选择**一个坐标轴**和该坐标轴上的一个实例作为**切分点**；(2) 确定**垂直于该坐标轴**且**过切分点**的超平面；(3) 将超矩形切分为左右两个子区域
3. 对空间进行递归，直到所有子区域中没有实例

操作中，我们往往使用实例点在坐标轴上的**中位数**作为切分点，这样可以得到**平衡的** kd 树。

具体的，在数据集上构造 kd 树算法如下：

输入：k 维空间数据集 $T = \{x_1, x_2, ..., x_N\}$，其中 $x_i = (x_i^1, x_i^2, ..., x_i^k)^T, i = 1, 2, ..., N$。

1. 构造根节点，根节点对应包含 T 的 k 为空间的超矩形区域
2. 选择 x^1 作为坐标轴，选取 $\{x_1^1, x_2^1, ..., x_N^1\}$ 的中位数作为切分点 (若中位数不在集合中，则使用离中位数最近且在集合中的数作为切分点)，将根节点对应的超矩形区域划分为两个子区域，将过切分平面的实例点存储在节点
3. 以 x^l 作为切分轴，其中 $l = j \mod k + 1, j$ 为该节点的深度，重复步骤 2，直至子区域均没有实例

输出：kd 树。

1.4.2 搜索 kd 树

以 kd 树的最近邻搜索为例，其搜索最邻近的方法是：

1. 先找到**包含目标点的叶节点**
2. 从该叶节点出发，**依次回退到父节点**，不断查找与目标点最邻近的节点，直到确定不存在更近的点为止

更为具体的，依次回退继续寻找最邻近的点的步骤是 (假设目前将回退到节点 A)：

1. 比较 A 中的节点是否比当前最近点更近，若更近则将其变为当前最近点
2. 以目标点为中心，目标点和当前最近点连线为半径做一个球体
3. 判断此球体是否与 A 的兄弟节点的超矩形区域相交
4. 若不相交，则回到 1 继续回退
5. 若相交，则在相交区域内寻找是否存在更近的点
6. 若不存在，则回到 1 继续回退
7. 若存在，则在找出更近的节点 B，将其设为最近点并继续回退

具体地，在 kd 树上搜索的步骤是：

输入：已构造好的 kd 树**和**目标点 x。

1. 在 kd 树中找到包含目标点 x 的叶节点：从根节点出发，递归地向下寻找。若 x 的当前维坐标小于切分点的坐标，则移动到左节点；否则移动到右节点
2. 以该叶节点为*当前最近点*
3. 递归地向上回退，在每个节点均执行以下操作：(1) 若该节点保存的实例比*当前最近点*更近，则以*当前节点*为*当前最近点*；(2) 检查**该节点的父节点的另一子节点**所对应的区域中是否与**以目标点为球心，目标点当前最近点为半径的球体**相交；(3) 如果不相交，则继续向上回退；(4) 如果相交，则在**该节点的父节点的另一子节点**所对应的区域中**递归**使用 kd 树最邻近搜索，找出该区域中 x 的最邻近点；(4) 如果该区域中的最邻近点比当前最近点远，则继续回退；(5) 如果该区域中的最邻近点比当前最近点近，则将该区域中的最邻近点设为当前最近点，并继续回退
4. 当退回到根节点时，搜索结束，存储的*当前最近点*即为 x 的最近邻

输出：x 的最近邻

2 代码实现

考虑以下数据:

$$T = \{(2,3)^T, (5,4)^T, (9,6)^T, (4,7)^T, (8,1)^T, (7,2)^T\}$$

其对应的标签依次为 0,1,2,3,4,5。目标点为 $(3,4.5)^T$。

2.1 sklearn 实现

准备数据：

```
import numpy as np
X = np.array([[2,3],[5,4],[9,6],[4,7],[8,1],[7,2]])
Y = np.array([i for i in range(6)]).reshape(-1,1)
```

k 邻近法的 sklearn 接口位于 sklearn.neighbors.KNeighborsClassifier, 其文档可见**此处**。其中较为重要的参数有：

- n_neighbors:k 的取值
- p:L_p 距离中的 p

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=1,p=2)
clf.fit(X,Y)
```

```
clf.predict(np.array([[3,4.5]]))
```

```
## array([0])
```

2.2 构造 kd 树

首先构造树中的节点，节点中需要储存的有：

- value: 该节点的实例
- idx: 切分坐标轴的下标
- left,right,father: 节点的左、右、父节点

```
class kdNode:
    def __init__(self):
        self.value = None
        self.idx = None
        self.father = None
        self.left = None
        self.right = None
```

定义一个 make_tree 函数，该函数可递归生成一棵 kd 树。

```
import math
def make_tree(node,data,idx):
    if len(data)>1:
        N,k = data.shape
        ## 寻找切分坐标轴和切分点
        idx = (idx+1)%k
        median = sorted(data[:,idx])[math.floor(N/2)]
        ## 从数据中找到切分实例
        median_id = list(data[:,idx]).index(median)
        value = data[median_id,:]
```

```
        ## 储存
        node.value = value
        node.idx = idx
        ## 切分剩余数据集
        data = np.delete(data,median_id,axis=0)
        left_data = data[data[:,idx]<median,:]
        right_data = data[data[:,idx]>=median,:]
        ## 递归左边和右边
        if len(left_data)!=0:
            node.left = kdNode()
            node.left.father = node
            make_tree(node.left,left_data,idx)
        if len(right_data)!=0:
            node.right = kdNode()
            node.right.father = node
            make_tree(node.right,right_data,idx)
    else:
        node.value = data[0]
```

调用该函数即可构建 kd 树。

```
class kdTree:
    def __init__(self,data):
        self.tree = kdNode()
        make_tree(self.tree,data,-1)
```

使用数据集 T 构建的 kd 树如下：

```
tree = kdTree(X)

def print_tree(tree,layer):
    print('\t'*layer,tree.value)
    if tree.left is not None:
        print_tree(tree.left,layer+1)
    if tree.right is not None:
        print_tree(tree.right,layer+1)

print_tree(tree.tree,0)
```

```
##  [7 2]
##    [5 4]
```

```
##       [2 3]
##       [4 7]
##   [9 6]
##       [8 1]
```

2.3 搜索 kd 树

定义 kdFInd 函数进行递归搜索：

```
def L2_dist(x,y):
    return np.sqrt(np.sum((x-y)**2))

def FindBrother(node):
    if node.father.left is node:
        return node.father.right
    if node.father.right is node:
        return node.father.left

def kdFind(node,x):
    ## 寻找叶节点
    while node.idx is not None:
        if x[node.idx]<node.value[node.idx]:
            if node.left is not None:
                node = node.left
            else:
                break
        else:
            if node.right is not None:
                node = node.right
            else:
                break
    ## 保存当前最近点
    nearest = node.value
    ## 不断向上回退直至根节点
    while node is not None and node.father is not None:
        ## 比较距离
        if L2_dist(node.value,x)<L2_dist(nearest,x):
            nearest = node.value
        ## 是否与兄弟节点相交
```

```
        dist = np.abs((node.value-x)[node.father.idx])
        r = L2_dist(nearest,x)
        ## 判断兄弟节点中有无更近的点
        if r>dist:
            try:
                parent_node = FindBrother(node)
                parent_node.father = None
                tmp = kdFind(parent_node,x)
                ## 存在，改动当前最近点
                if L2_dist(tmp,x)<L2_dist(nearest,x):
                    nearest = tmp
                ## 不存在，继续向上递归
                else:
                    node = node.father
                    continue
            except:
                pass
        node = node.father
    return nearest
```

对于点 $(3, 4.5)^T$，其最邻近点为：

```
kdFind(tree.tree,np.array([3,4.5]))
```

```
## array([2, 3])
```

朴素贝叶斯

1 朴素贝叶斯

1.1 后验概率

假设训练集为

$$T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$$

其中 $x_i \in R^n, y_i \in \{c_1, c_2, ..., c_K\}, i = 1, 2, ..., N$。

对于后验概率 $P(Y = c_k|X = x)$，可以由**贝叶斯定理**计算：

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

因此后验概率 $P(Y = c_k|X = x)$ 的计算等价于计算先验概率 $P(Y = c_k)$ 和条件概率 $P(X = x|Y = c_k)$。

1.2 条件独立性

朴素贝叶斯假定数据各属性间的**条件独立性**，即：

$$\begin{aligned} P(X = x|Y = c_k) &= P(X^1 = x^1, X^2 = x^2, ..., X^n = x^n|Y = c_k) \\ &= \prod_{j=1}^{n} P(X^j = x^j|Y = c_k) \end{aligned}$$

因此后验概率的计算又等价于先验概率 $P(Y = c_k)$ 和条件概率 $P(X^j = x^j|Y = c_k)$ 的计算。

1.3 极大似然估计

对于先验概率 $P(Y = c_k)$ 和条件概率 $P(X^j = x^j|Y = c_k)$，我们均可使用极大似然法来进行估计。两者的极大似然估计分别为：

$$P(Y=c_k)=\frac{\sum_{i=1}^{N}I(y_i=c_k)}{N},k=1,2,...,K$$

$$P(X^j=x^j|Y=c_k)=\frac{\sum_{i=1}^{N}I(x_i^j=x^j,y_i=c_k)}{\sum_{i=1}^{N}I(y_i=c_k)},j=1,2,...,n;k=1,2,...,K$$

1.4 朴素贝叶斯

朴素贝叶斯算法如下：

输入：数据集 $T=\{(x_1,y_1),(x_2,y_2),...,(x_N,y_N)\}$，其中 $x_i\in R^n,y_i\in\{c_1,c_2,...,c_K\},i=1,2,...,N$ 和实例 x。

1. 计算先验概率 $P(Y=c_k),k=1,2,...,K$

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

2. 计算条件概率 $P(X^j=x^j|Y=c_k),j=1,2,...,n;k=1,2,...,K$

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

3. 计算后验概率 $P(Y=c_k|X=x),k=1,2,...,K$

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

4. 确定 x 的类 $y=argmax_kP(Y=c_k|X=x)$

输出：x 的类 y。

1.5 拉普拉斯平滑

为防止出现极大似然估计出现估计概率为 0 的情况，常使用**平滑方式**，其方法如下：

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

$$P(X^j=x^j|Y=c_k)=\frac{\sum_{i=1}^{N}I(x_i^j=x^j,y_i=c_k)+\lambda}{\sum_{i=1}^{N}I(y_i=c_k)+S_j\lambda}$$

其中 S_j 为 X^j 的取值个数，$\lambda\geq 0$ 为平滑因子。当 $\lambda=1$ 时为**拉普拉斯平滑**。

2 代码实现

考虑以下数据：

X^1	X^2	Y
1	S	-1
1	M	-1
1	M	1
1	S	1
1	S	-1
2	S	-1
2	M	-1
2	M	1
2	L	1
2	L	1
3	L	1
3	M	1
3	M	1
3	L	1
3	L	-1

2.1 sklearn 实现

准备数据：

```
import numpy as np
X = np.array([[1,'S'],[1,'M'],[1,'M'],[1,'S'],[1,'S'],\
              [2,'S'],[2,'M'],[2,'M'],[2,'L'],[2,'L'],\
              [3,'L'],[3,'M'],[3,'M'],[3,'L'],[3,'L']])
Y = np.array([-1,-1,1,1,-1,-1,-1,1,1,1,1,1,1,1,-1]).reshape(-1,1)
```

朴素贝叶斯的接口位于 sklearn.naive_bayes 下，该类下有适用于离散型的 CategoricalNB 接口和连续型的 GaussianNB 接口等。此处使用 CategoricalNB 进行分类，其文档可见**此处**。其中较为重要的参数有：

- alpha: 平滑参数 λ

注意到 CategoricalNB 接口只接收数值型的输入，所以需要使用 sklearn.preprocessing.LabelEncoder 对其进行编码。

```
from sklearn.naive_bayes import CategoricalNB
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
le.fit(X[:,1])
```

```
X[:,1] = le.transform(X[:,1])
clf = CategoricalNB(alpha = 0)
clf.fit(X,Y)
```

对于新样本 $(2, S)^T$，预测结果如下：

```
new_x = np.array([[2,'S']])
new_x[:,1] = le.transform(new_x[:,1])
clf.predict_proba(new_x)
```

```
## array([[0.75, 0.25]])
```

2.2 朴素贝叶斯实现

定义一个 NaiveBayes 类进行实现。与 sklearn 中的接口类似，此处也定义了 fit 函数和 predict 函数分别进行拟合和预测。实现中为了便于实现，使用了 pandas 的计数功能。同时此处预测只能对单个新数据进行预测，事实上对代码稍加改动便可以实现同时预测多个新样本的功能。

```
import pandas as pd

class NaiveBayes:
    def __init__(self,lam=1):
        self.lam = lam

    def fit(self,X,Y):
        data = pd.concat([pd.DataFrame(X,columns=[str(i) for i in range(X.shape[1])]),\
        pd.DataFrame(Y,columns=['Y'])],axis=1)
        ## 计算先验概率
        self.piror = dict(data['Y'].value_counts())
        self.K = len(self.piror)
        ## 计算条件概率
        self.S = {}
        self.CondProb = {}
        for col in data.columns[:-1]:
            tmp = data[[col,'Y']]
```

```
            tmp = tmp.groupby([col,'Y']).agg({'Y':'count'})
            tmp.columns=['count']
            tmp = tmp.reset_index()
            self.CondProb[col] = tmp
            self.S[col] = len(tmp[col].value_counts())

    def predict(self,new_X):
        posterior = []
        for k in self.piror.keys():
            piror = (self.piror[k]+self.lam)/(sum(self.piror.values())+self.lam*self.K)
            for i in range(new_X.shape[1]):
                S = self.S[str(i)]
                cond = self.CondProb[str(i)]
                count = cond.loc[(cond[str(i)]==new_X[0,i])&(cond['Y']==k),['count']]
                count = (count.values[0,0]+self.lam)/(self.piror[k]+S*self.lam)
                piror *= count
            posterior.append(piror)
        posterior = [round(p/sum(posterior),2) for p in posterior]
        return dict(zip(self.piror.keys(),posterior))
```

其预测结果如下所示 (该段代码在 Jupyter 下可正常运行，但在 Rmd 中报错，经检查代码无错误)：

```
clf = NaiveBayes(lam=0)
clf.fit(X,Y)
clf.predict(np.array([[2,'S']]))
```

```
## {1: 0.25, -1: 0.75}
```

决策树

1 决策树

假设训练集为

$$D = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$$

其中，$x_i = (x_i^1, x_i^2, ..., x_i^n)^T$，n 为特征个数；$y_i \in \{1, 2, ..., K\}, i = 1, 2, ..., N$。

决策树每次要从特征 $x^1, x^2, ..., x^n$ 中选出一个特征，通过 if-else 判断语句将样本分类，使样本尽可能的分开。更一般地，决策树算法是递归地选择最优特征，并根据该特征对训练数据进行分割，使得对各个子数据集有一个最好的分类过程。

1.1 特征选择

定义 1：假设随机变量 X 的概率分布为 $P(X = x_i) = p_i, i = 1, 2, ..., n$，则随机变量 X 的**熵**定义为：

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

熵可以表现出随机变量的**不确定性**，熵越大，随机变量的不确定性就越大。

定义 2： **条件熵**$H(Y|X)$ 表示在已知随机变量 X 的条件下随机变量 Y 的不确定性。其定义为：

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

在计算熵时需要**注意**：

1. 通过样本计算的是经验熵和经验条件熵
2. 若有 0 概率，我们定义 $0 \log_2 0 = 0$

定义 3： **信息增益**表示得知特征 X 的信息后而使得类 Y 的信息的不确定性减少的程度。特征 X 对变量 Y 的信息增益 $g(Y,X)$ 定义为 Y 的经验熵 $H(Y)$ 和给定特征 X 条件下 Y 的条件经验熵 $H(Y|X)$ 之差，即：

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

易于发现，**信息增益大的特征具有更强的分类能力**。因此，基于信息增益的特征选择方法是每次计算所有特征的信息增益，并选择信息增益最大的特征。

定义 4： 特征 X 对变量 Y 的**信息增益比**$g_R(Y,X)$ 定义为其信息增益 $g(Y,X)$ 和Y **关于特征** X **的熵** $H_X(Y)$ 之比，即：

$$g_R(Y,X) = \frac{g(Y,X)}{H_X(Y)}$$

其中，$H_X(Y) = \sum_{i \in value(X)} \frac{|Y_i|}{|Y|} \log_2 \frac{|Y_i|}{|Y|}$，$n$ 为特征 X 的取值个数，$|Y_i|$ 为 Y 中其 X 取值为 x_i 的个数。

1.2 ID3 算法

ID3 算法在决策树的各个节点上**应用信息增益准则**选取特征，递归地构建决策树。

设 D 为训练数据。其有 K 个类 $C_k, k = 1,2,...,K, |C_k|$ 为属于类 C_k 的个数，$\sum_{i=1}^{K} |C_k| = |D|$。设特征 A 有 n 个不同的取值 $\{a_1, a_2, ..., a_n\}$，可以根据 A 的取值将 D 分为 n 个子集 $D_1, D_2, ..., D_n$，有 $\sum_{i=1}^{n} |D_i| = |D|$。记 D_{ik} 为集合 D_i 中属于 C_k 的样本，即 $D_{ik} = D_i \cap C_k$。

此时，特征 A 对数据集 D 的信息增益 $g(D,A)$ 为：

$$\begin{aligned} g(D,A) &= H(D) - H(D|A) \\ &= -\sum_{k=1}^{K} \frac{|C_k|}{|D|} \log_2 \frac{|C_k|}{|D|} - \sum_{i=1}^{n} \frac{|D_i|}{|D|} H(D_i) \\ &= -\sum_{k=1}^{K} \frac{|C_k|}{|D|} \log_2 \frac{|C_k|}{|D|} + \sum_{i=1}^{n} \frac{|D_i|}{|D|} \sum_{k=1}^{K} \frac{|D_{ik}|}{|D_k|} \log_2 \frac{|D_{ik}|}{|D_k|} \end{aligned}$$

在 ID3 算法中，需要多次计算信息增益。该算法具体如下：

输入： 训练数据集 D，特征集 A 阈值 ϵ。

1. 若 D 中所有实例属于同一类 C_k，则 T 为单节点树，并将 C_k 作为该节点的类标记
2. 若 $A = \emptyset$，则 T 为单节点树，并将 D 中实例数最大的类 C_k 作为该节点的类标记
3. 否则，对 A 中的各特征计算其**信息增益**，选择信息增益最大的特征 A_g

4. 如果特征 A_g 的信息增益小于 ϵ，则设置 T 为单节点树，并将该节点中实例数最大的类 C_k 作为该节点的类标记
5. 否则，对于 A_g 的每一可能值 a_i，将该节点的数据集分割为子集 D_i，建立子树 T_i。若 $D_i = \emptyset$，则将 D_i 的标记设置为该节点的标记，否则将 D_i 中实例数最大的类标记为 D_i 的类
6. 对第 i 个子节点，以 D_i 为训练集，$A - \{A_g\}$ 为特征集，**递归调用** 1-5

输出：决策树 T

1.3 C4.5 算法

C4.5 算法与 ID3 算法十分类似，唯一的区别是 C4.5 算法使用信息增益比进行特征选择。特征 A 对数据集 D 的信息增益比 $g_R(D, A)$ 为：

$$\begin{aligned} g_R(D, A) &= \frac{g(D, A)}{H_A(D)} \\ &= \frac{-\sum_{k=1}^{K} \frac{|C_k|}{|D|} \log_2 \frac{|C_k|}{|D|} + \sum_{i=1}^{n} \frac{|D_i|}{|D|} \sum_{k=1}^{K} \frac{|D_{ik}|}{|D_k|} \log_2 \frac{|D_{ik}|}{|D_k|}}{-\sum_{i=1}^{n} \frac{|D_i|}{|D|} \log_2 \frac{|D_i|}{|D|}} \end{aligned}$$

其算法如下：

输入：训练数据集 D，特征集 A 阈值 ϵ。

1. 若 D 中所有实例属于同一类 C_k，则 T 为单节点树，并将 C_k 作为该节点的类标记
2. 若 $A = \emptyset$，则 T 为单节点树，并将 D 中实例数最大的类 C_k 作为该节点的类标记
3. 否则，对 A 中的各特征计算其**信息增益比**，选择信息增益最大的特征 A_g
4. 如果特征 A_g 的信息增益小于 ϵ，则设置 T 为单节点树，并将该节点中实例数最大的类 C_k 作为该节点的类标记
5. 否则，对于 A_g 的每一可能值 a_i，将该节点的数据集分割为子集 D_i，建立子树 T_i。若 $D_i = \emptyset$，则将 D_i 的标记设置为该节点的标记，否则将 D_i 中实例数最大的类标记为 D_i 的类
6. 对第 i 个子节点，以 D_i 为训练集，$A - \{A_g\}$ 为特征集，**递归调用** 1-5

输出：决策树 T

1.4 剪枝

决策树无休止的生长会导致树的过拟合，因此需要简化生成的决策树，可对其进行**剪枝**。一种剪枝方式是将叶节点的个数以正则项加入到树的损失函数中。

假设决策树 T 的**叶节点个数**为 $|T|$。t 为树 T 的一个叶节点，该节点有 N_t 个样本点，其中 k 类的样本点有 N_{tk} 个,$k = 1, 2, ..., |T|$。记 $H_t(T)$ 为叶节点 t 上的经验熵，α 为一超参数，则可定义决策树的损失函数为：

$$
\begin{aligned}
C_\alpha(T) &= \sum_{t=1}^{|T|} N_t H_t(T) + \alpha|T| \\
&= \sum_{t=1}^{|T|} N_t(-\sum_{k=1}^{K} \frac{N_{tk}}{N_t} \log_2 \frac{N_{tk}}{N_t}) + \alpha|T| \\
&:= C(T) + \alpha|T|
\end{aligned}
$$

其中 $C(T)$ 相当于模型对数据的**训练误差**，$|T|$ 表示模型的复杂程度。通过 α 对模型的泛化能力进行调节。

在剪枝时，可以通过自下而上的方法递归实现，具体算法如下：

输入：决策树 T，超参数 α

1. 计算所有节点的经验熵
2. 递归地向上回缩。设一组叶节点回缩到父节点前后树分别为 T_B 和 T_A，如果 $C_\alpha(T_A) \leq C_\alpha(T_B)$，则进行剪枝。更一般地，假设该父节点为 P，其子节点分别为 $P_1, P_2, ..., P_n$，节点上的样本个数分别为 $N_1, N_2, ..., N_n$，且 $N = \sum_{i=1}^{n} N_i$。若 $N \cdot H(P) - \sum_{i=1}^{n} N_i H(P_i) \leq \alpha(n-1)$，则进行剪枝
3. 重复 2，直至不能继续

输出：修剪后的子树 T_α

1.5 CART 算法

与 ID3 算法和 C4.5 算法不同，CART 算法假定决策树是**二叉树**，且 CART 算法**既可以用于分类也可以用于回归**。当其用于回归时，使用**平方误差最小化**准则；当其用于分类时，使用**基尼指数最小化**准则。

1.5.1 回归树

决策树相当于将输入空间划分为多个单元，并将新样本分配到其中的某个单元上，将单元的输出值作为新样本的输出结果。假设输入空间被划分成了 M 个单元 $R_1, R_2, ..., R_M$，且在 R_m 上的输出值为 c_m，则回归树模型可表示为：

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

对于空间 R_m 上 c_m 的取值，根据平方误差最小化准则，可以得出 c_m 的最优值为 $\hat{c}_m = mean(y_i | x_i \in R_m)$。

对于决策树中节点的划分问题，相当于寻找切分变量 x^j 和切分点 s。两者会将空间切分成两个区域 $R_1(j,s) = \{x | x^j \leq s\}$ 和 $R_2(j,s) = \{x | x^j > s\}$。CART 算法通过最小化以下变量来确定 x^j 和 s:

$$\min_{j,s}[\min_{c_1}\sum_{x_i\in R_1(j,s)}(y_i-c_1)^2+\min_{c_2}\sum_{x_i\in R_2(j,s)}(y_i-c_2)^2]$$

对于固定的 j,s，代入 $c_1=mean(y_i|x_i\in R_1(j,s))$ 和 $c_2=mean(y_i|x_i\in R_2(j,s))$，可以得到最优的 j,s。不断地对节点进行该规则的划分即可得到一棵 (最小二乘) 回归树。其算法如下：

输入：训练集数据 D。

1. 遍历切分变量 j 和切分点 s，求解

$$\min_{j,s}[\min_{c_1}\sum_{x_i\in R_1(j,s)}(y_i-c_1)^2+\min_{c_2}\sum_{x_i\in R_2(j,s)}(y_i-c_2)^2]$$

2. 使用 j,s 将节点划分为 $R_1(j,s)=\{x|x^j\leq s\}$ 和 $R_2(j,s)=\{x|x^j>s\}$，并设置 $c_i=\frac{1}{|R_m(j,s)|}\sum_{x\in R_m(j,s)}y_1, m=1,2$
3. 对两个子区域**递归**调用 1,2，直至满足停止条件
4. 生成决策树 $f(x)=\sum_{m=1}^{M}c_mI(x\in R_m)$

输出：回归树 $f(x)$。

1.5.2 分类树

分类树则使用基尼指数选择最优特征和其切分点。

定义 5：假设随机变量 X 的概率分布为 $P(X=x_k)=p_k, k=1,2,...,K$，则随机变量 X 的**基尼指数**定义为：

$$Gini(X)=\sum_{k=1}^{K}p_k(1-p_k)=1-\sum_{k=1}^{K}p_k^2$$

基尼指数越大，样本集合的不确定性也就越大。对于数据集 D，其基尼指数为 $Gini(D)=1-\sum_{k=1}^{K}(\frac{|C_k|}{|D|})^2$。

定义 6：如果根据特征 A 按 $A=a$ 将 D 划分为两个部分 $D_1=x\in D|A(x)=a$ 和 $D_2=x\in D|A(x)\neq a$，则在特征 A 条件下的集合 D 的基尼指数 $Gini(D,A)$ 定义为：

$$Gini(D,A)=\frac{|D_1|}{|D|}Gini(D_1)+\frac{|D_2|}{|D|}Gini(D_2)$$

使用这一指数可以得到 CART 算法：

输入：训练数据 D 和停止条件。

1. 遍历所有特征 A 和其可能的取值 a，根据条件 $A=a$ 将样本空间分为两部分，计算基尼指数 $Gini(D,A=a)$

2. 选择最小的基尼指数对应的特征 A 和对应的切分点 a 对数据集进行切分，生成两个子节点
3. 对两个子节点**递归**调用 1,2，直到满足停止条件
4. 生成 CART 决策树

输出：CART 决策树。

1.5.3 剪枝

对于任意树 T，其剪枝时的损失函数为 $C_\alpha(T)=C(T)+\alpha|T|$。$\alpha$ 的取值决定了树的复杂程度。

对于树中的任意一个节点 t，假设其子树为 T_t。以 t 为单节点的树的损失函数为 $C_\alpha(t)=C(t)+\alpha$，而以 t 为根节点的子树 T_t 的损失函数为 $C_\alpha(T_t)=C(T_t)+\alpha|T_t|$。

显然，当 $\alpha\to 0^+$ 时，$C_\alpha(T_t)<C_\alpha(t)$；当 $\alpha\to+\infty$ 时，$C_\alpha(T_t)>C_\alpha(t)$。因此存在某一 α，使得 $C_\alpha(T_t)=C_\alpha(t)$。事实上，取 $\alpha=\frac{C(t)-C(T_t)}{|T_t|-1}$ 时即可满足条件。在这种情况下，**单节点 t 和以 t 为根节点的树 T_t 拥有相同的损失**，我们倾向于选择更简单的树 t。

特别地，定义 $g(t)=\frac{C(t)-C(T_t)}{|T_t|-1}$。$g(t)$ 的大小就表示了对节点 t 剪枝后整体损失函数的减小程度。显然 $g(t)$ 越小越好，所以对于任意树 T，我们的目标即为找到 T 中 $g(t)$ 最小的节点 t。

此外我们可以证明：存在 $0=\alpha_0<\alpha_1<...<\alpha_n<+\infty$，当 $\alpha\in[\alpha_i,\alpha_{i+1}),i=0,1,...,n$，剪枝得到的子树序列 $\{T_0,T_1,...,T_n\}$ 是嵌套的。这说明：在计算 $g(t)$ 时，我们无需对每个节点单独计算，而是可以自上而下的进行计算。我们可以从根节点开始向下计算，得到最优的剪枝子树列 $\{T_0,T_1,...,T_n\}$ 以及对应的参数 $\{\alpha_0,\alpha_1,...,\alpha_n\}$。随后我们便可以通过交叉验证法从 $\{T_0,T_1,...,T_n\}$ 选出最优的决策树 T_α。注意，此处 α 不是超参数，而是通过交叉验证法自我习得的。

更一般地，CART 剪枝算法如下：

输入：CART 算法生成的决策树 T_0。

1. 记 $k=0,T=T_0,\alpha=+\infty$
2. 自上而下逐层对决策树中的每个节点 t 计算 $C(T_t),|T_t|,g(t)=\frac{C(t)-C(T_t)}{|T_t|-1}$ 和 $\alpha=\min(\alpha,g(t))$，其中 $C(T_t)$ 是对训练数据的预测误差
3. 对 $g(t)=\alpha$ 的内部节点进行剪枝，对节点 t 以多数表决法决定其类，得到树 T
4. $k=k+1,\alpha_k=\alpha,T_k=T$
5. 重复操作步骤 2-4 直至 T_k 仅包含根节点
6. 采用交叉验证法从树序列 $T_0,T_1,...,T_n$ 中选取最优子树 T_α

2 代码实现

考虑的数据集为贷款申请样本数据表，具体可见 excel 文档。

2.1 sklearn 实现

决策树的 sklearn 接口位于 sklearn.tree.DecisionTreeClassifier, 其文档可见**此处**。

注意： sklearn 中的接口使用的是 CART 的改进算法，同时也不支持属性变量 (scikit-learn uses an optimised version of the CART algorithm; however, scikit-learn implementation does not support categorical variables for now.)

sklearn 中的接口也不提供剪枝功能,提高泛化能力通过调节 max_depth,min_samples_split,min_samples_leaf 等参数实现。

准备数据：

```
import pandas as pd
data = pd.read_excel('data.xlsx')
X = data[['年龄', '有工作', '有自己的房子', '信贷情况']].values
Y = data[['类别']].values
data
```

```
##     年龄 有工作 有自己的房子 信贷情况  类别
## 0   青年   否       否   一般   0
## 1   青年   否       否    好   0
## 2   青年   是       否    好   1
## 3   青年   是       是   一般   1
## 4   青年   否       否   一般   0
## 5   中年   否       否   一般   0
## 6   中年   否       否    好   0
## 7   中年   是       是    好   1
## 8   中年   否       是  非常好   1
## 9   中年   否       是  非常好   1
## 10  老年   否       是  非常好   1
## 11  老年   否       是    好   1
## 12  老年   是       否    好   1
## 13  老年   是       否  非常好   1
## 14  老年   否       否   一般   0
```

同样，我们需要用 LabelEncoder 对属性变量编码。

```
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier

le_tran = [LabelEncoder() for _ in range(4)]
for i in range(4):
```

```
    le_tran[i].fit(X[:,i])
    X[:,i] = le_tran[i].transform(X[:,i])
```

```
clf = DecisionTreeClassifier()
clf.fit(X,Y)
```

其生成的决策树如下：

```
from sklearn.tree import export_text
tree_representation = export_text(clf,feature_names=list(data.columns[:-1]))
print(tree_representation)
```

```
## |--- 有自己的房子 <= 0.50
## |   |--- 有工作 <= 0.50
## |   |   |--- class: 0
## |   |--- 有工作 >  0.50
## |   |   |--- class: 1
## |--- 有自己的房子 >  0.50
## |   |--- class: 1
```

2.2 ID3 和 C4.5 决策树

决策树的节点定义如下，其主要存储如下信息：

- feature: 该节点将要对哪个特征进行分支
- value: 该节点的变量分布，如 { ：6, ：9}
- label: 该节点的标签
- entropy: 该节点的熵
- subnode: 以字典形式存储该节点的子节点

```
class DTNode:

    def __init__(self):
        self.feature = None
        self.value = {}
        self.label = None
        self.entropy = None
        self.subnode = {}
```

决策树可如下表示，类内主要的函数有：

- init 函数：初始化时，需给定 method(ID3 or C4.5) 和阈值 eps

- fit 函数：拟合决策树，其中使用了 build_tree 函数进行递归
- print_tree 函数：可视化决策树，其中使用 print_node 函数进行递归
- predict 函数：进行预测，其中使用 predict_sample 函数以可同时对多个样本同时进行预测
- purn 函数：对决策树剪枝，其中使用 cut_node 函数进行递归

```
import numpy as np
import math

class DecisionTree:

    def __init__(self,method,eps):
        self.root = None
        self.method = method
        self.eps = eps

    def compute_entropy(self,value):
        value = [v/sum(value) for v in value]
        value = [v for v in value if v!=0]
        value = [-v*math.log(v) for v in value]
        return sum(value)

    def fit(self,X,Y):
        self.var_num = X.shape[1]
        df = pd.concat([pd.DataFrame(X,columns = ['var'+str(i) for i in range(self.var_num)]),\
                  pd.DataFrame(Y,columns=['y'])],axis=1)
        self.root = DTNode()
        self.bulid_tree(self.root,df,method=self.method,eps=self.eps)


    def bulid_tree(self,node,df,method,eps=0):
        node.value = dict(df['y'].value_counts())

        ## 均属于一个类
        if len(node.value) == 1:
            node.label = list(node.value.keys())[0]
            node.entropy = 0
        ## 无特征
        elif df.shape[1] == 1:
            node.label = [key for key,value in node.value.items() \
```

```
                if value == max(node.value.values())][0]
            node.entropy = self.compute_entropy(list(node.value.values()))

        ## 需递归
        else:
            node.label = [key for key,value in node.value.items() \
                if value == max(node.value.values())][0]
            node.entropy = self.compute_entropy(list(node.value.values()))
            ### 计算特征的信息增益
            var_list = list(df.columns[:-1])
            info = {}
            if method == 'ID3':
                for var in var_list:
                    df_tmp = [l[1] for l in list(df[[var,'y']].groupby(var))]
                    cond_info = sum([self.compute_entropy(list(d['y'].value_counts()))*len(d)\
                         for d in df_tmp])/len(df)
                    info[var] = node.entropy-cond_info
            elif method == 'C4.5':
                for var in var_list:
                    df_tmp = [l[1] for l in list(df[[var,'y']].groupby(var))]
                    cond_info = sum([self.compute_entropy(list(d['y'].value_counts()))*len(d)\
                         for d in df_tmp])/len(df)
                    info[var] = (node.entropy-cond_info)/ \
                        self.compute_entropy(list(df[var].value_counts()))
            ### 选出特征
            max_info = max(list(info.values()))
            feature = [key for key in info.keys() if info[key]==max_info][0]

            if max_info>=eps:
                node.feature = feature

                ### 建立子树
                fea_space = list(np.unique(df[feature].values))
                new_col = [l for l in list(df.columns) if l!=feature]
                for fea in fea_space:
                    node.subnode[fea] = DTNode()
                    self.bulid_tree(node.subnode[fea],df.loc[df[node.feature]==fea,new_col],\
                                    method=method,eps=eps)
```

```
    def print_node(self,node,var_dict,layer=0):
        if len(node.subnode)==0:
            print('|'+'\t|'*layer+'---类别:'+str(node.label)+', 分布:'+str(node.value))
        else:
            for key,value in node.subnode.items():
                print('|'+'\t|'*layer+'---'+var_dict[node.feature]+':'+str(key))
                self.print_node(value,var_dict,layer+1)

    def print_tree(self,col_name=None):
        var_name = ['var'+str(i) for i in range(self.var_num)]
        if col_name is not None:
            var_dict = dict(zip(var_name,col_name))
        else:
            var_dict = dict(zip(var_name,var_dict))

        self.print_node(self.root,var_dict,layer=0)

    def predict_sample(self,new_X):
        node = self.root
        while len(node.subnode)!=0:
            try:
                node = node.subnode[new_X[int(node.feature[3])]]
            except:
                break
        return node.label

    def predict(self,new_X):
        return np.apply_along_axis(self.predict_sample,axis=1,arr=new_X)

    def cut_node(self,node,alpha):
        subnode_entropy = sum([sum(n.value.values())*n.entropy for n in node.subnode.values()])
        if subnode_entropy!=0:
            for subnode in node.subnode.values():
                if subnode.entropy!=0:
                    self.cut_node(subnode,alpha)

        subnode_entropy = sum([sum(n.value.values())*n.entropy for n in node.subnode.values()])
        if subnode_entropy!=0:
```

```
            delta = sum(node.value.values())*node.entropy-subnode_entropy
            if delta<=alpha*(len(node.subnode)-1):
                node.subnode={}
                node.feature = None

    def prun(self,alpha):
        self.cut_node(self.root,alpha)
```

对数据进行拟合和可视化结果如下：

```
clf = DecisionTree(method='ID3',eps=0)
clf.fit(X,Y)
clf.print_tree(col_name=list(data.columns[:-1]))
```

```
## |---有自己的房子:否
## |    |---有工作:否
## |    |   |---类别:0,分布:{0: 6}
## |    |---有工作:是
## |    |   |---类别:1,分布:{1: 3}
## |---有自己的房子:是
## |    |---类别:1,分布:{1: 6}
```

对新数据预测结果如下：

```
new_X = np.array([['中年', '是', '否', '好'],\
                  ['青年', '否', '否', '非常好']])
clf.predict(new_X)
```

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

2.3 剪枝

对该决策树剪枝结果如下：

```
clf.prun(alpha=5)
clf.print_tree(col_name=list(data.columns[:-1]))
```

```
## |---类别:1,分布:{1: 9, 0: 6}
```

注意，该决策树不存在过拟合现象，因此需要给定一个较大的 α 值才会进行剪枝。且实践证明，当树的第二层被剪枝时，此时的超参数 α 也会对第一层的树进行剪枝。

2.4 CART 决策树

此处主要实现回归树，使用的数据集是汽车售价数据，数据如下：

```
import numpy as np
import pandas as pd
data = pd.read_csv('cars.csv')
data['type'] = data['type'].apply(lambda x:{'small':0,'midsize':1,'large':2}[x])
X = data[['type', 'price', 'mpg_city', 'passengers']].values
Y = data[['weight']].values
data.head()
```

```
##    type  price  mpg_city  passengers  weight
## 0     0   15.9        25           5    2705
## 1     1   33.9        18           5    3560
## 2     1   37.7        19           6    3405
## 3     1   30.0        22           4    3640
## 4     1   15.7        22           6    2880
```

CART 决策树的节点需要存储如下信息：

- cut_var：该节点的切分变量
- cut_point：该节点的切分点
- avg：该节点内样本的平均值
- depth：该节点的深度
- num：该节点的样本个数
- left,right：该节点的左子节点和右子节点

```
class CARTNode:

    def __init__(self):
        self.cut_var = None
        self.cut_point = None
        self.avg = None
        self.depth = None
        self.num = None
        self.left = None
        self.right = None
```

回归树内如下：

```
class RegressionTree:

    def __init__(self,max_depth=float('inf'),min_samples_leaf=1):
        self.max_depth = max_depth
        self.min_samples_leaf = min_samples_leaf

    def compute_loss(self,df_var,df_y,s):
        df = pd.concat([df_var,df_y],axis=1)
        df.columns = ['x','y']
        df_1 = df[df['x']<=s]
        df_2 = df[df['x']>s]
        c_1 = np.mean(df_1['y'])
        c_2 = np.mean(df_2['y'])
        loss = np.sum((df_1['y']-c_1).values**2)+np.sum((df_2['y']-c_2).values**2)
        return loss

    def fit(self,X,Y):
        self.var_num = X.shape[1]
        df = pd.concat([pd.DataFrame(X,columns = ['var'+str(i) for i in range(self.var_num)]),\
            pd.DataFrame(Y,columns = ['y'])],axis=1)
        self.root = CARTNode()
        self.build_tree(self.root,df)

    def build_tree(self,node,df,depth=0):
        node.avg = np.mean(df['y'])
        node.num = len(df)
        node.depth = depth

        if node.depth < self.max_depth and node.num > self.min_samples_leaf:

            ## 寻找切分变量和切分点
            cut = []
            for j in range(self.var_num):
                s_list = sorted(np.unique(df['var'+str(j)]))[:-1]
                for s in s_list:
                    loss = self.compute_loss(df[['var'+str(j)]],df[['y']],s)
                    cut.append([[j,s],loss])
```

```
            loss = [c[1] for c in cut]
            min_loss = min(loss)
            cut_var, cut_point = [c[0] for c in cut][loss.index(min_loss)]

            node.cut_var = 'var'+str(cut_var)
            node.cut_point = cut_point

            ## 递归
            node.left = CARTNode()
            self.build_tree(node.left,df[df[node.cut_var]<=node.cut_point],depth+1)
            node.right = CARTNode()
            self.build_tree(node.right,df[df[node.cut_var]>node.cut_point],depth+1)

    def print_node(self,node,var_dict,layer=0):
        if node.left is None and node.right is None:
            print('|'+'\t|'*layer+'---输出:'+str(round(node.avg,3))+', 样本个数:'+str(node.num))

        if node.left is not None:
            print('|'+'\t|'*layer+'---'+var_dict[node.cut_var]+'<='+str(node.cut_point))
            self.print_node(node.left,var_dict,layer+1)

        if node.right is not None:
            print('|'+'\t|'*layer+'---'+var_dict[node.cut_var]+'>'+str(node.cut_point))
            self.print_node(node.right,var_dict,layer+1)


    def print_tree(self,col_name=None):
        var_name = ['var'+str(i) for i in range(self.var_num)]
        if col_name is not None:
            var_dict = dict(zip(var_name,col_name))
        else:
            var_dict = dict(zip(var_name,var_dict))

        self.print_node(self.root,var_dict,layer=0)
```

假定高度最大为 4，叶节点内的样本个数最少为 5，回归树的结果如下：

```
clf = RegressionTree(max_depth=4,min_samples_leaf=5)
clf.fit(X,Y)
```

```
clf.print_tree(col_name=list(data.columns[:-1]))
```

```
## |---type<=0.0
## |    |---mpg_city<=29.0
## |    |   |---price<=9.2
## |    |   |   |---输出:2295.0,样本个数:4
## |    |   |---price>9.2
## |    |   |   |---mpg_city<=25.0
## |    |   |   |   |---输出:2603.0,样本个数:5
## |    |   |   |---mpg_city>25.0
## |    |   |   |   |---输出:2414.0,样本个数:5
## |    |---mpg_city>29.0
## |    |   |---price<=8.6
## |    |   |   |---输出:1887.5,样本个数:4
## |    |   |---price>8.6
## |    |   |   |---输出:2251.667,样本个数:3
## |---type>0.0
## |    |---mpg_city<=18.0
## |    |   |---price<=35.2
## |    |   |   |---price<=23.7
## |    |   |   |   |---输出:3988.333,样本个数:3
## |    |   |   |---price>23.7
## |    |   |   |   |---输出:3605.0,样本个数:6
## |    |   |---price>35.2
## |    |   |   |---输出:3996.667,样本个数:3
## |    |---mpg_city>18.0
## |    |   |---price<=18.2
## |    |   |   |---mpg_city<=19.0
## |    |   |   |   |---输出:3610.0,样本个数:1
## |    |   |   |---mpg_city>19.0
## |    |   |   |   |---输出:2993.333,样本个数:6
## |    |   |---price>18.2
## |    |   |   |---price<=26.7
## |    |   |   |   |---输出:3415.5,样本个数:10
## |    |   |   |---price>26.7
## |    |   |   |   |---输出:3535.0,样本个数:4
```

支持向量机

1 原始问题和对偶问题

1.1 原始问题

支持向量机中需要使用原始问题和对偶问题的转换来简化问题的求解。

假设 $f(x), c_i(x), h_j(x)$ 是定义在 R^n 上的连续可微函数。将如下带约束的最优化问题称为原始问题：

$$\begin{aligned} \min_{x \in R^n} \quad & f(x) \\ s.t. \quad & c_i(x) \le 0, i = 1, 2, ..., k \\ & h_j(x) = 0, j = 1, 2, ..., l \end{aligned}$$

对其引入广义拉格朗日函数：

$$L(x, \alpha, \beta) = f(x) + \sum_{i=1}^{k} \alpha_i c_i(x) + \sum_{j=1}^{l} \beta_j h_j(x)$$

其中，α_i, β_j 是拉格朗日乘子，$\alpha_i \ge 0$。考虑以下函数：

$$\begin{aligned} \theta_P(x) &= \max_{\alpha, \beta : \alpha_i \ge 0} L(x, \alpha, \beta) \\ &= \max_{\alpha, \beta : \alpha_i \ge 0} [f(x) + \sum_{i=1}^{k} \alpha_i c_i(x) + \sum_{j=1}^{l} \beta_j h_j(x)] \end{aligned}$$

当 x 满足约束条件 $c_i(x) \le 0$ 和 $h_j(x) = 0$ 时，有：

$$\begin{aligned} L(x, \alpha, \beta) &= f(x) + \sum_{i=1}^{k} \alpha_i c_i(x) + \sum_{j=1}^{l} \beta_j h_j(x) \\ &\le f(x) + \sum_{i=1}^{k} \alpha_i \cdot 0 + \sum_{j=1}^{l} \beta_j \cdot 0 \\ &= f(x) \end{aligned}$$

而当 x 不满足约束条件时，则当 $c_i(x) > 0$ 时，令 $\alpha_i \to +\infty$，当 $h_j(x) \neq 0$ 时，令 $\beta_j \to \infty \cdot I(h_j(x) < 0)$，其余的 α_i, β_j 均取为 0。此时有：

$$\max_{\alpha,\beta:\alpha_i \geq 0} [f(x) + \sum_{i=1}^{k} \alpha_i c_i(x) + \sum_{j=1}^{l} \beta_j h_j(x)] = +\infty$$

因此，

$$\theta_P(x) = \max_{\alpha,\beta:\alpha_i \geq 0} L(x, \alpha, \beta) = f(x)$$

且取到最大值时，x 必位于约束条件区域内。所以**求解原始问题的最小值等价于求解拉格朗日函数的极小极大问题**：

$$\min_x \max_{\alpha,\beta:\alpha_i \geq 0} L(x, \alpha, \beta)$$

此外不妨假设原始问题的最优解 $p^* = \min_x \theta_P(x)$。

1.2 对偶问题

原始问题求解的是 $\min_x \max_{\alpha,\beta:\alpha_i \geq 0} L(x, \alpha, \beta)$，而对偶问题求解的则是 $\max_{\alpha,\beta:\alpha_i \geq 0} \min_x L(x, \alpha, \beta)$，可以将其表示为如下约束最优化问题：

$$\begin{aligned} &\max_{\alpha,\beta} \theta_D(\alpha, \beta) := \max_{\alpha,\beta:\alpha_i \geq 0} \min_x L(x, \alpha, \beta) \\ &s.t. \quad \alpha_i \geq 0, i = 1, 2, ..., k \end{aligned}$$

不妨假设对偶问题的最优解 $d^* = \max_{\alpha,\beta:\alpha_i \geq 0} \theta_D(\alpha, \beta)$。

1.3 原始问题和对偶问题

我们可以简单地证明：若原始问题和对偶问题都有最优值，则 $d^* \leq p^*$。

Proof：

$$\min_x L(x, \alpha, \beta) \leq L(x, \alpha, \beta) \leq \max_{\alpha,\beta:\alpha_i \geq 0} L(x, \alpha, \beta)$$

上式对任意满足条件的 x, α, β 均成立，因此有：

$$\max_{\alpha,\beta:\alpha_i\geq 0}\min_x L(x,\alpha,\beta)\leq L(x,\alpha,\beta)\leq \min_x\max_{\alpha,\beta:\alpha_i\geq 0} L(x,\alpha,\beta)$$

因此 $d^*\leq p^*$。

以上结果说明：原始问题和对偶问题是相关的，但不一定等价。

首先，若 x^* 和 α^*,β^* 分别是原始问题和对偶问题的可行解且 $d^*=p^*$，则 x^* 和 α^*,β^* 分别是原问题和对偶问题的最优解。因此，当 $d^*=p^*$ 时，我们便可以使用解对偶问题来解原始问题。

其次，**对于支持向量机问题，其原始问题和对偶问题是等价的。**

再者，在满足一些条件的情况下（如在支持向量机中），原始问题和对偶问题的解 x^* 和 α^*,β^* 和其满足 KKT 条件互为充分必要条件。KKT 条件为：

1. $\nabla_x L(x^*,\alpha^*,\beta^*)=0$
2. $\alpha_i^* c_i^*(x^*)=0, i=1,2,...,k$
3. $c_i^*(x^*)\leq 0, i=1,2,...,k$
4. $\alpha_i^*\geq 0, i=1,2,...,k$
5. $h_j(x^*)=0, j=1,2,...,l$

2 线性支持向量机

2.1 硬间隔

给定数据集 $T=\{(x_1,y_1),(x_2,y_2),...,(x_N,y_N)\}, x_i\in R^n, y_i\in\{-1,1\}, i=1,2,...,N$。此处数据集 T 线性可分。

由于 T 线性可分，所以存在无穷个超平面可将数据正确分开。硬间隔支持向量机 (线性可分支持向量机) 将使用**间隔最大化**从这些超平面中选出最优的分离超平面。

定义：对于给定的数据集 T 和超平面 (ω,b)。超平面 (ω,b) 关于样本点 (x_i,y_i) 的几何间隔为：

$$\gamma_i=y_i(\frac{\omega}{||\omega||}\cdot x_i+\frac{b}{||\omega||})$$

易于发现在分类正确的情况下，γ_i 即为点 (x_i,y_i) 至超平面 (ω,b) 的距离，且 γ_i 的取值与 ω,b 的数量级无关。记 $\gamma=\min_{i=1,2,...,N}\gamma_i$。

硬间隔支持向量机最大化几何间隔，因此其可表示为以下优化问题：

$$\begin{aligned} \max_{\omega,b} \quad & \gamma \\ s.t. \quad & y_i(\frac{\omega}{||\omega||} \cdot x_i + \frac{b}{||\omega||}) \geq \gamma, i = 1, 2, ..., N \end{aligned}$$

令 $\hat{\gamma} = ||\omega||\gamma$，该优化问题便等价于：

$$\begin{aligned} \max_{\omega,b} \quad & \frac{\hat{\gamma}}{||\omega||} \\ s.t. \quad & y_i(\omega \cdot x_i + b) \geq \hat{\gamma}, i = 1, 2, ..., N \end{aligned}$$

此外，由于 ω, b 的同比例放缩不影响该问题的求解，不妨设 $\hat{\gamma} = 1$。此外由于最大化 $\frac{1}{||\omega||}$ 和最小化 $\frac{1}{2}||\omega||^2$ 等价。因此该问题又等价于：

$$\begin{aligned} \min_{\omega,b} \quad & \frac{1}{2}||\omega||^2 \\ s.t. \quad & y_i(\omega \cdot x_i + b) - 1 \geq 0, i = 1, 2, ..., N \end{aligned}$$

这是一个凸二次规划问题。因此求解线性可分支持向量机便等价于求解以上优化问题的最优解 ω^*, b^*。

2.2 对偶问题

对于支持向量机的优化问题，其对偶问题比原始问题更容易求解。引入拉格朗日乘子 $\alpha_i \geq 0, i = 1, 2, ..., N$，该优化问题的拉格朗日函数为：

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

对偶问题优化如下表达式：$\max_\alpha \min_{\omega,b} L(\omega, b, \alpha)$，首先求 $\min_{\omega,b} L(\omega, b, \alpha)$。

$$\begin{aligned} \frac{\partial L(\omega, b, \alpha)}{\partial \omega} &= \omega - \sum_{i=1}^{N} \alpha_i y_i x_i \\ \frac{\partial L(\omega, b, \alpha)}{\partial b} &= -\sum_{i=1}^{N} \alpha_i y_i \end{aligned}$$

令两式等于 0，可得 $\omega = \sum_{i=1}^{N} \alpha_i y_i x_i, \sum_{i=1}^{N} \alpha_i y_i = 0$。将其代回拉格朗日函数有：

$$\begin{aligned}\min_{\omega,b} L(\omega,b,\alpha) &= \min_{\omega,b} \frac{1}{2}||\omega||^2 + \sum_{i=1}^{N}\alpha_i(1-y_i(\omega\cdot x_i+b)) \\ &= \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(x_i\cdot x_j) + \sum_{i=1}^{N}\alpha_i - \sum_{i=1}^{N}\alpha_i y_i((\sum_{j=1}^{N}\alpha_j y_j x_j)\cdot x_i + b) \\ &= -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(x_i\cdot x_j) + \sum_{i=1}^{N}\alpha_i\end{aligned}$$

因此对偶问题等价于对 α 最大化 $\min_{\omega,b} L(\omega,b,\alpha)$，这等价于以下最优化问题：

$$\begin{aligned}\min_{\alpha} \quad & \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(x_i\cdot x_j) - \sum_{i=1}^{N}\alpha_i \\ s.t. \quad & \sum_{i=1}^{N}\alpha_i y_i = 0 \\ & \alpha_i \geq 0, \quad i=1,2,...,N\end{aligned}$$

该对偶问题和原始问题**相互等价**。因此可以先求出对偶问题的最优解 α^*，再将 α^* 转化为原始问题的最优解 ω^*, b^*。

由 KKT 条件，可知 $\nabla_\omega L(\omega^*,b^*,\alpha^*) = \omega^* - \sum_{i=1}^{N}\alpha_i^* y_i x_i = 0$，因此有 $\omega^* = \sum_{i=1}^{N}\alpha_i^* y_i x_i$。

此外可以注意到，至少有一个 $\alpha_j^* > 0$(否则有 $\omega^* = 0$，其不为原始问题的解)。由 KKT 条件知，对 j 有：$y_j(\omega^*\cdot x_j + b^*) - 1 = 0$，因此：

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

因此硬间隔支持向量机算法可如下：

输入：线性可分的数据集 $T=\{(x_1,y_1),(x_2,y_2),...,(x_N,y_N)\}$，其中 $x_i\in R^n, y_i\in\{-1,1\}, i=1,2,...,N$。

1. 求解约束最优化问题：

$$\begin{aligned}\min_{\alpha} \quad & \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(x_i\cdot x_j) - \sum_{i=1}^{N}\alpha_i \\ s.t. \quad & \sum_{i=1}^{N}\alpha_i y_i = 0 \\ & \alpha_i \geq 0, \quad i=1,2,...,N\end{aligned}$$

2. 得到问题的最优解 $\alpha^* = (\alpha_1^*,\alpha_2^*,...,\alpha_N^*)^T$，计算：

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

同时选择 α^* 的一个正分量 $\alpha_j^* > 0$，计算：

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

3. 得到分离超平面 $\omega^* \cdot x + b^* = 0$ 和分类决策函数 $f(x) = sign(\omega^* \cdot x + b^*)$

输出：分离超平面和分类决策函数。

2.3 软间隔

硬间隔支持向量机要求数据集线性可分。不过通常情况下，数据集存在一些特异点，导致数据集不是线性可分的。但在去掉了这些点后，数据集仍是线性可分的。

线性不可分意味着数据集中的某些样本点 (x_i, y_i) 不能满足条件 $y_i(\omega \cdot x_i + b) - 1 \geq 0$。我们可以加入一个松弛变量 $\xi_i \geq 0$，降低约束条件的要求，使约束条件变为：

$$y_i(\omega \cdot x_i + b) \geq 1 - \xi_i$$

不过为了防止约束条件变得过于松弛，我们需要在目标函数上添加关于 ξ_i 的正则项。引入惩罚超参数 C，目标函数可以调整为 $\frac{1}{2}||\omega||^2 + C\sum_{i=1}^{N} \xi_i$。因此，软间隔支持向量机可以转化以下优化问题：

$$\begin{aligned} \min_{\omega,b} \quad & \frac{1}{2}||\omega||^2 + C\sum_{i=1}^{N} \xi_i \\ s.t. \quad & y_i(\omega \cdot x_i + b) \geq 1 - \xi_i, \quad i = 1, 2, ..., N \\ & \xi_i \geq 0, \quad i = 1, 2, ..., N \end{aligned}$$

可以证明该优化问题中：ω 的解是唯一的，但 b 的解可能不唯一。

同样地，我们尝试将软间隔支持向量机也转换为对偶问题。引入乘子 $\alpha_i \geq 0, \mu_i \geq 0$，该问题的拉格朗日函数是：

$$L(\omega, b, \xi, \alpha, \mu) = \frac{1}{2}||\omega||^2 + C\sum_{i=1}^{N} \xi_i + \sum_{i=1}^{N} \alpha_i (1 - \xi_i - y_i(\omega \cdot x_i + b)) - \sum_{i=1}^{N} \mu_i \xi_i$$

因此，对偶问题便是求解以下极大极小问题：

$$\max_{\alpha,\mu} \min_{\omega,b,\xi} L(\omega, b, \xi, \alpha, \mu)$$

首先求解：$\min_{\omega,b,\xi} L(\omega, b, \xi, \alpha, \mu)$。对变量分别求偏导有：

$$\frac{\partial L(\omega,b,\xi,\alpha,\mu)}{\partial \omega}=\omega-\sum_{i=1}^{N}\alpha_i y_i x_i=0$$
$$\frac{\partial L(\omega,b,\xi,\alpha,\mu)}{\partial b}=-\sum_{i=1}^{N}\alpha_i y_i=0$$
$$\frac{\partial L(\omega,b,\xi,\alpha,\mu)}{\partial \xi_i}=C-\alpha_i-\mu_i=0$$

可以求得：

$$\omega=\sum_{i=1}^{N}\alpha_i y_i x_i$$
$$\sum_{i=1}^{N}\alpha_i y_i=0$$
$$C-\alpha_i-\mu_i=0$$

将其代回拉格朗日函数，有：

$$\begin{aligned}\min_{\omega,b,\xi} L(\omega,b,\xi,\alpha,\mu)&=\min_{\omega,b,\xi}[\frac{1}{2}||\omega||^2+C\sum_{i=1}^{N}\xi_i+\sum_{i=1}^{N}\alpha_i(1-\xi_i-y_i(\omega\cdot x_i+b))-\sum_{i=1}^{N}\mu_i\xi_i]\\&=-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(x_i\cdot x_j)+\sum_{i=1}^{N}(C-\alpha_i-\mu_i)\xi_i-\sum_{i=1}^{N}\alpha_i y_i b+\sum_{i=1}^{N}\alpha_i\\&=-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(x_i\cdot x_j)+\sum_{i=1}^{N}\alpha_i\end{aligned}$$

因此，对偶问题可转化为如下问题：

$$\begin{aligned}\max_{\alpha,\mu}\quad&-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(x_i\cdot x_j)+\sum_{i=1}^{N}\alpha_i\\ s.t.\quad&\sum_{i=1}^{N}\alpha_i y_i=0\\&C-\alpha_i-\mu_i=0\\&\alpha_i\geq 0\\&\mu_i\geq 0\end{aligned}$$

注意到后三个约束条件等价于 $0\leq\alpha_i\leq C$。所以，该对偶问题可以简化为如下形式：

$$\begin{aligned}\min_{\alpha} \quad & \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j (x_i \cdot x_j) - \sum_{i=1}^{N}\alpha_i \\ s.t. \quad & \sum_{i=1}^{N}\alpha_i y_i = 0 \\ & 0 \le \alpha_i \le C, \quad i = 1, 2, ..., N\end{aligned}$$

同理，我们可以通过 KKT 条件，在求得 α 的最优解 α^* 后，可如下求得 ω^*, b^*：

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

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

其中，下标 j 对应的 α_j^* 有：$0 \le \alpha_j^* \le C$。注意到，由于满足条件的 α_j^* 可能有许多个，所以 b^* 不唯一。

软间隔支持向量机算法可总结如下：

输入：数据集 $T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$，其中 $x_i \in R^n, y_i \in \{-1, 1\}, i = 1, 2, ..., N$ 和惩罚超参数 C。

1. 求解约束最优化问题：

$$\begin{aligned}\min_{\alpha} \quad & \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j (x_i \cdot x_j) - \sum_{i=1}^{N}\alpha_i \\ s.t. \quad & \sum_{i=1}^{N}\alpha_i y_i = 0 \\ & 0 \le \alpha_i \le C, \quad i = 1, 2, ..., N\end{aligned}$$

2. 得到问题的最优解 $\alpha^* = (\alpha_1^*, \alpha_2^*, ..., \alpha_N^*)^T$，计算：

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

同时选择 α^* 的一个分量满足 $0 \le \alpha_j^* \le C$，计算：

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

3. 得到分离超平面 $\omega^* \cdot x + b^* = 0$ 和分类决策函数 $f(x) = sign(\omega^* \cdot x + b^*)$

输出：分离超平面和分类决策函数。

2.4 合页损失

软间隔支持向量机的原始问题为：

$$\begin{aligned}\min_{\omega,b} \quad & \frac{1}{2}||\omega||^2 + C\sum_{i=1}^{N}\xi_i \\ s.t. \quad & y_i(\omega\cdot x_i + b) \geq 1-\xi_i, \quad i=1,2,...,N \\ & \xi_i \geq 0, \quad i=1,2,...,N\end{aligned}$$

令

$$\xi_i = [1-y_i(\omega\cdot x_i+b)]_+ = \begin{cases} 1-y_i(\omega\cdot x_i+b) & ,1-y_i(\omega\cdot x_i+b)>0 \\ 0 & ,1-y_i(\omega\cdot x_i+b)\leq 0 \end{cases}$$

所以 $\xi_i \geq 0$。且当 $1-y_i(\omega\cdot x_i+b)>0$ 时，有 $y_i(\omega\cdot x_i+b)=1-\xi_i$；当 $1-y_i(\omega\cdot x_i+b)\leq 0$ 时，有 $y_i(\omega\cdot x_i+b)\geq 1=1-\xi_i$。综上即有 $y_i(\omega\cdot x_i+b)\geq 1-\xi_i$。因此 ω, b, ξ_i 满足原始问题的约束条件，若取 $\lambda=\frac{1}{2C}$，目标函数可转化为

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

该目标函数中的第一项称为**合页损失函数**。因此支持向量机的优化问题也可转为合页损失函数的优化问题。不过合页损失函数不是连续可导的，优化较为困难，从而使用也不太广泛。

3 非线性支持向量机

3.1 核技巧

对于非线性的分类问题，可以使用核技巧使用非线性支持向量机对数据进行分类。对于非线性问题，往往采用的方法是使用一个非线性变换，将非线性问题转换为线性问题，并通过求解变换后的线性问题来求解原始的非线性问题。核技巧的基本想法就是通过一个非线性变换将输入空间映射到特征空间，并在特征空间中求解线性支持向量机问题。

定义：设 X 是输入空间，H 是特征空间，如果存在一个从 X 到 H 的映射 $\phi(x): X\to H$，使得对于任意 $x, z\in X$，函数 $K(x,z)$ 满足：

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

则称 $K(x,z)$ 是核函数，$\phi(x)$ 是映射函数。

核技巧巧妙的地方在于：我们**可以在只知道核函数 $K(x,z)$ 而不知道映射函数 $\phi(x)$ 的情况下求解问题**。

因此，核技巧的使用便等价于核函数的选取。一般来说，构造核函数时一般使用正定核或 Mercer 核。常用的核函数有：

- 线性核函数：$K(x,z)=x\cdot z$
- 多项式核函数：$K(x,z)=(x\cdot z+1)^p$
- 高斯核函数：$K(x,z)=\exp(-\frac{||x-z||^2}{2\sigma^2})$

3.2 非线性支持向量机

线性支持向量机的对偶问题为：

$$\begin{aligned}\min_{\alpha}\quad & \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(x_i\cdot x_j)-\sum_{i=1}^{N}\alpha_i\\ s.t.\quad & \sum_{i=1}^{N}\alpha_i y_i=0\\ & 0\le\alpha_i\le C,\quad i=1,2,...,N\end{aligned}$$

但使用核技巧时，其目标函数便可转换为：

$$\begin{aligned}\theta(\alpha)&=\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(\phi(x_i)\cdot\phi(x_j))-\sum_{i=1}^{N}\alpha_i\\&=\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j K(x_i,x_j)-\sum_{i=1}^{N}\alpha_i\end{aligned}$$

可以看到目标函数只与核函数 $K(.,.)$ 有关，映射函数 $\phi(.)$ 在计算中被消去了。同样，对于预测函数中的映射函数也可用核函数进行代替：

$$\begin{aligned}f(x)&=sign(\omega^*\cdot\phi(x)+b^*)\\&=sign((\sum_{i=1}^{N}\alpha_i^* y_i\phi(x_i))\cdot\phi(x)+b^*)\\&=sign(\sum_{i=1}^{N}\alpha_i^* y_i K(x_i,x)+b^*)\end{aligned}$$

因此，非线性支持向量机便可在只给出核函数的情况下进行求解。其算法步骤如下：

输入：数据集 $T=\{(x_1,y_1),(x_2,y_2),...,(x_N,y_N)\}$，其中 $x_i\in R^n, y_i\in\{-1,1\}, i=1,2,...,N$，核函数 $K(x,z)$ 和惩罚超参数 C。

1. 求解约束最优化问题：

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j K(x_i,x_j) - \sum_{i=1}^{N}\alpha_i \\ s.t. \quad & \sum_{i=1}^{N}\alpha_i y_i = 0 \\ & 0 \le \alpha_i \le C, \quad i = 1,2,...,N \end{aligned}$$

2. 得到问题的最优解 $\alpha^* = (\alpha_1^*, \alpha_2^*, ..., \alpha_N^*)^T$，选择 α^* 的一个分量满足 $0 \le \alpha_j^* \le C$，计算：

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

3. 得到分类决策函数 $f(x) = sign(\sum_{i=1}^{N}\alpha_i^* y_i K(x_i, x) + b^*)$

输出：分类决策函数。

4 优化算法：SMO 算法

目前支持向量机中还有一个问题需要解决，即如何求解支持向量机的优化问题。SMO 算法是一种高效解决支持向量机学习问题的优化方法。SMO 算法可以求解如下凸二次规划的对偶问题：

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j K(x_i,x_j) - \sum_{i=1}^{N}\alpha_i \\ s.t. \quad & \sum_{i=1}^{N}\alpha_i y_i = 0 \\ & 0 \le \alpha_i \le C, \quad i = 1,2,...,N \end{aligned}$$

SMO 算法从 KKT 条件入手。由于 KKT 条件是该优化问题的充要条件，因此 SMO 算法每次选出 2 个变量进行子问题的优化，其中一个变量是违反 KKT 条件最严重的一个。如此，SMO 算法的步骤便可分解为如下两个步骤：

1. 从所有变量中选出两个变量
2. 对这两个变量构造一个子二次规划问题，并对其进行优化 (该问题存在解析解)

SMO 算法不断重复以上两个步骤，直至收敛。

4.1 子二次规划求解

不妨设选择的两个变量是 α_1, α_2，其余变量 $\alpha_3, \alpha_4, ..., \alpha_N$ 在此次规划问题被视作固定的。因此，在子问题中，略去常数项，SMO 算法的目标函数可以表示为：

$$\begin{aligned}\min_{\alpha_1,\alpha_2} \quad & W(\alpha_1,\alpha_2)=\frac{1}{2}K_{11}\alpha_1^2+\frac{1}{2}K_{22}\alpha_2^2+y_1y_2K_{12}\alpha_1\alpha_2- \\ & (\alpha_1+\alpha_2)+y_1\alpha_1\sum_{i=3}^{N}y_i\alpha_iK_{i1}+y_2\alpha_2\sum_{i=3}^{N}y_i\alpha_iK_{i2} \\ s.t. \quad & \alpha_1y_1+\alpha_2y_2=-\sum_{i=3}^{N}\alpha_iy_i:=\zeta \\ & 0\le\alpha_i\le C, \quad i=1,2\end{aligned}$$

其中，$K_{ij}=K(x_i,x_j),i,j=1,2,...,N,\zeta$ 是常数。记：

$$g(x)=\sum_{i=1}^{N}\alpha_iy_iK(x_i,x)+b$$

$$E_i:=g(x_i)-y_i=\sum_{j=1}^{N}\alpha_jy_jK(x_j,x_i)+b-y_i, \quad i=1,2$$

$$v_i:=\sum_{j=3}^{N}\alpha_jy_jK(x_i,x_j)=g(x_i)-\sum_{j=1}^{2}\alpha_jy_jK(x_i,x_j)-b, \quad i=1,2$$

因此目标函数可以写成：

$$\begin{aligned}W(\alpha_1,\alpha_2)=&\frac{1}{2}K_{11}\alpha_1^2+\frac{1}{2}K_{22}\alpha_2^2+y_1y_2K_{12}\alpha_1\alpha_2- \\ &(\alpha_1+\alpha_2)+y_1v_1\alpha_1+y_2v_2\alpha_2\end{aligned}$$

由于 $\alpha_1=\frac{1}{y_1}(\zeta-\alpha_2y_2)=y_1(\zeta-\alpha_2y_2)$。将其代入目标函数 $W(\alpha_1,\alpha_2)$ 中，其可变为一个单变量问题：

$$\begin{aligned}W(\alpha_2)=&\frac{1}{2}K_{11}(\zeta-\alpha_2y_2)^2+\frac{1}{2}K_{22}\alpha_2^2+y_2K_{12}(\zeta-\alpha_2y_2)\alpha_2- \\ &(\zeta-\alpha_2y_2)y_1-\alpha_2+v1(\zeta-\alpha_2y_2)+y_2v_2\alpha_2\end{aligned}$$

对 α_2 求导，有：

$$\begin{aligned}\frac{\partial W(\alpha_2)}{\partial\alpha_2}=&K_{11}\alpha_2-K_{11}\zeta y_2+K_{22}\alpha_2-2K_{12}\alpha_2+K_{12}\zeta y_2+ \\ &y_1y_2-1-v_1y_2+y_2v_2\end{aligned}$$

令其等于 0，可以得到：

$$
\begin{aligned}
(K_{11}+K_{22}-2K_{12})\alpha_2 &= y_2(y_2-y_1+\zeta K_{11}-\zeta K_{12}+v_1-v_2) \\
&= y_2[y_2-y_1+\zeta K_{11}-\zeta K_{12}+(g(x_1)-\sum_{j=1}^{2}y_j\alpha_j K_{1j}-b)- \\
&\quad (g(x_2)-\sum_{j=1}^{2}y_j\alpha_j K_{2j}-b)]
\end{aligned}
$$

等号右侧将 $\zeta=\alpha_1 y_1+\alpha_2 y_2$ 代入，经过整理。等号右侧表达式等价于 $(K_{11}+K_{22}-2K_{12})\alpha_2+y_2(E_1-E_2)$。因此，若令 $\eta=K_{11}+K_{22}-2K_{12}$，参数更新表达式为：

$$
\alpha_2^{new}:=\alpha_2^{old}+\frac{y_2(E_1-E_2)}{\eta}
$$

不过由于该优化问题有 $0\le\alpha_1,\alpha_2\le C$ 的约束条件，因此参数的更新还需要经过剪辑。易知，参数 α_2^{new} 有下界 L 和上界 H，满足 $L\le\alpha_2^{new}\le H$，其中：

$$
L=\begin{cases}\max(0,\alpha_2^{old}-\alpha_1^{old}) & ,y_1\ne y_2\\ \max(0,\alpha_2^{old}+\alpha_1^{old}-C) & ,y_1=y_2\end{cases},
H=\begin{cases}\min(C,C+\alpha_2^{old}-\alpha_1^{old}) & ,y_1\ne y_2\\ \min(C,\alpha_2^{old}+\alpha_1^{old}) & ,y_1=y_2\end{cases}
$$

因此，α_2 的更新表达式需改变为：

$$
\alpha_2^{new}=\begin{cases}H & ,\alpha_2^{old}+\frac{y_2(E_1-E_2)}{\eta}>H\\ \alpha_2^{old}+\frac{y_2(E_1-E_2)}{\eta} & ,L\le\alpha_2^{old}+\frac{y_2(E_1-E_2)}{\eta}\le H\\ L & ,\alpha_2^{old}+\frac{y_2(E_1-E_2)}{\eta}<L\end{cases}
$$

此时，α_1 的更新公式为：

$$
\alpha_1^{new}=\alpha_1^{old}+y_1y_2(\alpha_2^{old}-\alpha_2^{new})
$$

4.2 变量选择方法

SMO 算法在子问题中只选择两个变量进行优化，其中一个是违反 KKT 条件的。

4.2.1 违反 KKT 条件的变量选取

训练样本应满足 KKT 条件，即：

$$\begin{aligned} \alpha_i = 0 &\iff y_i g(x_i) \geq 1 \\ 0 < \alpha_i < C &\iff y_i g(x_i) = 1 \\ \alpha_i = C &\iff y_i g(x_i) \leq 1 \end{aligned}$$

检验时先从所有 $0 < \alpha_i < C$ 的点上开始寻找，随后再找其他点，偏差最大的点可成为第一个选取的变量

4.2.2 第二个变量的选择

一个简单的做法是选择使 $|E_1 - E_2|$ 最大的变量，即如果 $E_1 > 0$，则选择最小的 E_i 作为 E_2，否则选择最大的。不过若此方法选出的两个变量对目标函数的下降趋势帮助不大，则可尝试先重新选择第二个变量，再重新选择第一个变量的启发式方法。

4.2.3 更新 b 和 E_i

由于在计算 α 时需要使用到 b 和 E_i，因此需不断对其进行更新。

当 $0 < \alpha_1^{new} < C$ 时，由 KKT 条件知：

$$\sum_{i=1}^{N} \alpha_i y_i K_{i1} + b = y_1$$

所以，

$$b_1^{new} = y_1 - \sum_{i=3}^{N} \alpha_i y_i K_{i1} - \alpha_1^{new} y_1 K_{11} - \alpha_2^{new} y_2 K_{21}$$

此外，由于

$$E_1 = \sum_{i=3}^{N} \alpha_i y_i K_{i1} + \alpha_1^{old} y_1 K_{11} + \alpha_2^{old} y_2 K_{21} + b^{old} - y_1$$

代入，有

$$b_1^{new} = b^{old} - E_1 - y_1 K_{11}(\alpha_1^{new} - \alpha_1^{old}) - y_2 K_{21}(\alpha_2^{new} - \alpha_2^{old})$$

同理，如果 $0 < \alpha_2^{new} < C$，则

$$b_2^{new} = b^{old} - E_2 - y_1 K_{12}(\alpha_1^{new} - \alpha_1^{old}) - y_2 K_{22}(\alpha_2^{new} - \alpha_2^{old})$$

如果 $\alpha_1^{new}, \alpha_2^{new}$ 同时满足 $0 < \alpha_i^{new} < C, i = 1, 2$，则 $b_1^{new} = b_2^{new}$。如果 $\alpha_1^{new}, \alpha_2^{new}$ 中有 0 或 C，则设置 $b^{new} = \frac{b_1^{new} + b_2^{new}}{2}$。

E_i 的更新方法如下：

$$E_i^{new} = \sum_{\{j:\alpha_j \leq C\}} y_j \alpha_j K(x_i, x_j) + b^{new} - y_i$$

综上，SMO 算法如下：

输入：数据集 $T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$，其中 $x_i \in R^n, y_i \in \{-1, 1\}, i = 1, 2, ..., N$ 和精度 ϵ。

1. 取初值 $\alpha^{(0)} = 0, b = 0$，令 $k = 0$
2. 选取优化变量 $\alpha_1^{(k)}, \alpha_2^{(k)}$
3. 计算变量 E_1, E_2, η
4. 更新变量 $\alpha_1^{(k)}, \alpha_2^{(k)}$ 至 $\alpha_1^{(k+1)}, \alpha_2^{(k+1)}$
5. 更新变量 E_1, E_2, b
6. 若在精度 ϵ 内满足停止条件：

$$y_i \cdot g(x_i) \begin{cases} \geq 1 & ,\{x_i | \alpha_i = 0\} \\ = 1 & ,\{x_i | 0 < \alpha_i < C\} \\ \leq 1 & ,\{x_i | \alpha_i = C\} \end{cases}$$

则令 $\alpha = \alpha^{(k+1)}$ 并输出，否则令 $k = k + 1$ 并回到 2

输出：α^*。

5 代码实现

考虑的数据集为西瓜 *3.0* 数据表，具体可见 txt 文档。

5.1 sklearn 实现

准备数据：

```
import pandas as pd
data = pd.read_table('melon.txt',sep=' ')
data['好瓜'] = data['好瓜'].apply(lambda x:2*x-1)
```

```
X = data[['密度','含糖率']].values
Y = data['好瓜'].values
```

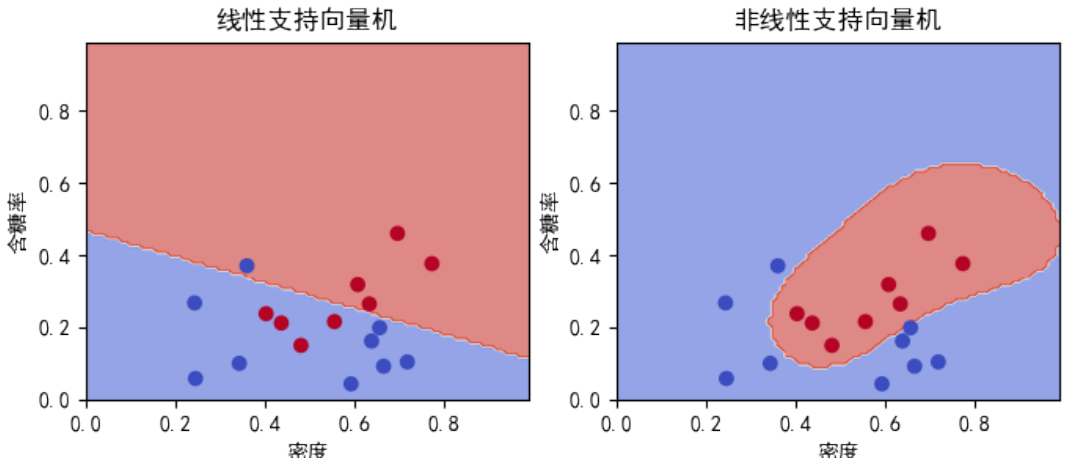
支持向量机的 sklearn 接口位于 sklearn.svm.SVC, 其文档可见**此处**。其中较为重要的参数有：

- C：惩罚项
- kernel：核函数，选项包括：linear，poly，rbf 等

```
from sklearn.svm import SVC
clf_linear = SVC(C=5,kernel='linear')
clf_linear.fit(X,Y)
clf_rbf = SVC(C=5,kernel='rbf')
clf_rbf.fit(X,Y)
```

画出对应的分割平面，可以看出非线性支持向量机在此问题中比线性支持向量机更适合。

```
import matplotlib.pyplot as plt
plt.rcParams['font.sans-serif']=['SimHei']
plt.rcParams['axes.unicode_minus']=False

plt.figure(figsize=(8,4),dpi=100)
title = ['线性支持向量机','非线性支持向量机']
for i,clf in enumerate([clf_linear,clf_rbf]):
    plt.subplot(1,2,i+1)
    xx, yy = np.meshgrid(np.arange(0,1,0.01),np.arange(0,1,0.01))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.6)
    plt.scatter(X[:,0], X[:,1], c=Y, cmap=plt.cm.coolwarm)
    plt.xlabel('密度')
    plt.ylabel('含糖率')
    plt.title(title[i])
```



5.2 SMO 算法和核支持向量机

我们使用 SMO 算法来实现支持向量机，此处只实现其一个简化版的 SMO 算法。该简化算法总体上与原始的 SMO 算法类似，仅仅是在选取变量时有微小的改动，对第二个变量使用随机筛选法，具体可见此处。

```
class SVM:

    def __init__(self,kernel='linear',d=3,sigma=1,
                      C=1,tol=1e-2,max_pass=50):
        self.C = C
        self.tol = tol
        self.max_passes = max_pass

        if kernel=='linear':
            def kernel(x1,x2):
                return np.sum(x1*x2)
        elif kernel=='poly':
            def kernel(x1,x2):
                return np.sum((x1*x2+1)**d)
        elif kernel=='rbf':
            def kernel(x1,x2):
                return np.exp(-np.sum((x1-x2)**2)/(2*sigma**2))

        self.kernel = kernel

    def fit(self,X,Y):
```

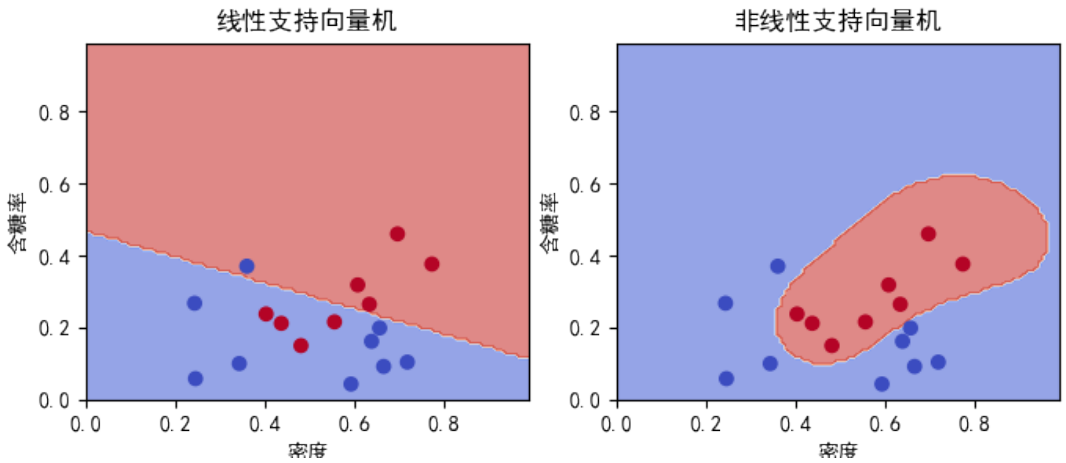
```
        N = X.shape[0]
        self.N = N
        self.alpha = np.zeros(N)
        self.b = 0
        passes = 0
        self.X = X
        self.Y = Y

        self.K = np.zeros((N,N))
        for i in range(N):
            for j in range(N):
                self.K[i,j] = self.kernel(X[i,:],X[j,:])

        while passes<self.max_passes:
            num_changed_alphas = 0
            for i in range(N):
                E_i = np.sum(self.alpha*Y*self.K[i,:])+self.b-Y[i]
                if (Y[i]*E_i<-self.tol and self.alpha[i]<self.C) or\
                   (Y[i]*E_i>self.tol and self.alpha[i]>0):
                    j = np.random.choice([j for j in range(N) if j!=i])
                    E_j = np.sum(self.alpha*Y*self.K[j,:])+self.b-Y[j]
                    alpha_i_old,alpha_j_old = self.alpha[i],self.alpha[j]
                    if Y[i] != Y[j]:
                        L = max(0,self.alpha[j]-self.alpha[i])
                        H = min(self.C,self.C+self.alpha[j]-self.alpha[i])
                    else:
                        L = max(0,self.alpha[i]+self.alpha[j]-self.C)
                        H = min(self.C,self.alpha[i]+self.alpha[j])

                    if L==H:
                        continue
                    eta = 2*self.K[i,j]-self.K[i,i]-self.K[j,j]
                    if eta>=0:
                        continue

                    alpha_j = self.alpha[j]-Y[j]*(E_i-E_j)/eta
                    alpha_j = H if alpha_j>H else alpha_j
                    alpha_j = L if alpha_j<L else alpha_j
```

```
                    if np.abs(alpha_j-self.alpha[j])<1e-5:
                        continue

                    alpha_i = self.alpha[i]+Y[i]*Y[j]*(alpha_j_old-alpha_j)

                    b1 = self.b-E_i-Y[i]*(alpha_i-alpha_i_old)*self.K[i,i]\
                                -Y[j]*(alpha_j-alpha_j_old)*self.K[i,j]
                    b2 = self.b-E_j-Y[i]*(alpha_i-alpha_i_old)*self.K[i,j]\
                                -Y[j]*(alpha_j-alpha_j_old)*self.K[j,j]

                    if 0<alpha_i and alpha_i<self.C:
                        b = b1
                    elif 0<alpha_j and alpha_j<self.C:
                        b = b2
                    else:
                        b = (b1+b2)/2

                    self.alpha[i],self.alpha[j]=alpha_i,alpha_j
                    self.b = b
                    num_changed_alphas += 1

            if num_changed_alphas==0:
                passes += 1
            else:
                passes = 0

    def predict_sample(self,new_X):
        return np.sign(np.sum([self.alpha[i]*self.Y[i]*self.kernel(self.X[i,:],new_X)\
            for i in range(self.N)])+self.b)

    def predict(self,new_X):
        return [self.predict_sample(new_X[i,:]) for i in range(new_X.shape[0])]
```

其结果如下：

```
import matplotlib.pyplot as plt
plt.rcParams['font.sans-serif']=['SimHei']
plt.rcParams['axes.unicode_minus']=False
```

```
clf_linear = SVM(C=5)
clf_linear.fit(X,Y)
clf_rbf = SVM(kernel='rbf',C=5,sigma=0.2)
clf_rbf.fit(X,Y)

plt.figure(figsize=(8,3),dpi=100)
title = ['线性支持向量机','非线性支持向量机']
for i,clf in enumerate([clf_linear,clf_rbf]):
    plt.subplot(1,2,i+1)
    xx, yy = np.meshgrid(np.arange(0,1,0.01),np.arange(0,1,0.01))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = np.array(Z).reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.6)
    plt.scatter(X[:, 0], X[:, 1], c=Y, cmap=plt.cm.coolwarm)
    plt.xlabel('密度')
    plt.ylabel('含糖率')
    plt.title(title[i])
```



AdaBoost

1 AdaBoost

1.1 提升方法

提升算法是从弱学习算法出发，反复学习，得到一系列弱分类器（也称基本分类器），然后组合这些弱分类器，构成一个强分类器。其中最重要的两个问题是：

- 如何改变训练数据的权值或概率分布
- 如何将弱分类器组合成强分类器

1.2 AdaBoost 算法

AdaBoost 是一种提升算法。在训练时，AdaBoost 提高那些被前一轮弱分类器错误分类样本的权值，以改变训练数据的权值；此外，AdaBoost 采取加权多数表决的方法，即加大分类误差率小的弱分类器的权值。

给定数据集

$$T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$$

其中，$x_i \in R^n, y_i \in \{-1, 1\}, i = 1, 2, ..., N$。AdaBoost 算法如下：

输入：数据集 $T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$，其中，$x_i \in R^n, y_i \in \{-1, 1\}, i = 1, 2, ..., N$，弱分类器个数 M **和**弱学习算法。

1. 初始化训练数据的权值分布

$$D_1 = (w_{11}, w_{12}, ..., w_{1N}), w_{1i} = \frac{1}{N}, i = 1, 2, ..., N$$

2. 对 $m = 1, 2, ..., M$:

- 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 $G_m(x) : X \to \{-1, 1\}$
- 计算 $G_m(x)$ 在训练集数据上的分类误差率

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

- 计算 $G_m(x)$ 的系数

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

- 更新训练集数据的权值分布

$$D_{m+1} = (w_{m+1,1}, w_{m+1,2}, ..., w_{m+1,N})$$

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

其中，Z_m 是规范化因子：$Z_m = \sum_{i=1}^{N}\exp(-\alpha_m y_i G_m(x_i))$

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

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

输出：分类器 $G(x)$。

1.3 训练误差分析

AdaBoost 算法的最终训练误差为 $\frac{1}{N}\sum_{i=1}^{N} I(G(x_i) \neq y_i)$。由于当 $G(x_i) \neq y_i$ 时，$y_i f(x_i) < 0$，因此 $\exp(-y_i f(x_i)) \geq 1$，因此有：

$$\begin{aligned}
\frac{1}{N}\sum_{i=1}^{N} I(G(x_i) \neq y_i) &\leq \frac{1}{N}\sum_{i=1}^{N}\exp(-y_i f(x_i)) \\
&= \frac{1}{N}\sum_{i=1}^{N}\exp(-\sum_{m=1}^{M}\alpha_i y_i G_m(X_i)) \\
&= \sum_{i=1}^{N} w_{1i}\prod_{m=1}^{M}\exp(-\alpha_i y_i G_m(X_i)) \\
&= Z_1\sum_{i=1}^{N} w_{2i}\prod_{m=2}^{M}\exp(-\alpha_i y_i G_m(X_i)) \\
&= Z_1 Z_2\sum_{i=1}^{N} w_{3i}\prod_{m=3}^{M}\exp(-\alpha_i y_i G_m(X_i)) \\
&= ... \\
&= Z_1 Z_2 ... Z_{M-1}\sum_{i=1}^{N} w_{Mi}\exp(-\alpha_M y_i G_M(X_i)) \\
&= \prod_{m=1}^{M} Z_m
\end{aligned}$$

这说明，在每一轮可以选取适当的 G_m 使得 Z_m 最小，从而使训练误差下降最快。此外，由于：

$$
\begin{aligned}
Z_m &= \sum_{i=1}^{N} w_{mi} \exp(-\alpha_i y_i G_m(X_i)) \\
&= \sum_{y_i = G_m(x_i)} w_{mi} e^{-\alpha_m} + \sum_{y_i \neq G_m(x_i)} w_{mi} e^{\alpha_m} \\
&= (1 - e_m) e^{-\alpha_m} + e_m e^{\alpha_m}
\end{aligned}
$$

将 $\alpha_m = \frac{1}{2} \ln \frac{1-e_m}{e_m}$ 代入，有：

$$
Z_m = 2\sqrt{e_m(1-e_m)} = \sqrt{1 - 4\gamma_m^2} \leq \exp(-2\gamma_m^2)
$$

其中 $\gamma_m = \frac{1}{2} - e_m$。式子中的不等式可以由 $\sqrt{1-x} \leq e^x, x \geq 0$ 得出。所以若存在 $\gamma > 0$，对所有 m 有 $\gamma_m \geq \gamma$，则有：

$$
\frac{1}{N} \sum_{i=1}^{N} I(G(x_i) \neq y_i) = \prod_{m=1}^{M} Z_m \leq \exp(-2 \sum_{m=1}^{M} \gamma_m^2) \leq \exp(-2M\gamma^2)
$$

因此，AdaBoost 的训练误差是以指数速率下降的。

2 代码实现

考虑以下数据：

x	y
0	1
1	1
2	1
3	-1
4	-1
5	-1
6	1
7	1
8	1
9	-1

2.1 sklearn 实现

准备数据：

```
import numpy as np
X = np.array([i for i in range(10)]).reshape(-1,1)
Y = np.array([1,1,1,-1,-1,-1,1,1,1,-1])
```

AdaBoost 的 sklearn 接口位于 sklearn.ensemble.AdaBoostClassifier, 其文档可见**此处**。其中较为重要的参数有：

- base_estimator：基分类器，默认为深度为 1 的决策树
- n_estimators：基分类器的个数，默认为 50

```
from sklearn.ensemble import AdaBoostClassifier
clf = [AdaBoostClassifier(n_estimators=i+1) for i in range(4)]
for c in clf:
    c.fit(X,Y)
```

```
for c in clf:
    print('基分类器个数：',clf.index(c)+1,'准确率：',np.sum(c.predict(X)==Y)/len(Y))
```

```
## 基分类器个数： 1 准确率： 0.7
## 基分类器个数： 2 准确率： 0.9
## 基分类器个数： 3 准确率： 1.0
## 基分类器个数： 4 准确率： 1.0
```

2.2 AdaBoost 实现

此处 AdaBoost 也使用深度为 1 的决策树作为基分类器。

```
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
X = np.array([i for i in range(10)]).reshape(-1,1)
Y = np.array([1,1,1,-1,-1,-1,1,1,1,-1]).reshape(-1,1)
```

AdaBoost 类如下：

```
class AdaBoost:

    def __init__(self,n_estimators=10):
        self.n_estimators = n_estimators
```

```
    def fit(self,X,Y):
        self.clf_list = []
        self.alpha_list = []

        weight = [1/len(Y) for _ in range(len(Y))]
        df = pd.concat([pd.DataFrame(X),pd.DataFrame(Y,columns=['y']),\
                        pd.DataFrame({'weight':weight})],axis=1)

        for m in range(self.n_estimators):
            X = df.iloc[:,:-2].values
            Y = df.iloc[:,-2].values
            weight = df.iloc[:,-1].values.reshape(-1)

            clf = DecisionTreeClassifier(max_depth=1)
            clf.fit(X,Y,sample_weight = weight)

            df['predict'] = list(clf.predict(X))
            err = np.sum(df['predict']!=df['y'])/len(df)
            alpha = np.log((1-err)/err)/2

            df['weight'] = df['weight']*np.exp(-alpha*df['y']*df['predict'])
            df['weight'] = df['weight']/np.sum(df['weight'])
            del df['predict']

            self.clf_list.append(clf)
            self.alpha_list.append(alpha)

    def predict(self,new_X):
        tmp = np.array([clf.predict(new_X) for clf in self.clf_list])
        tmp = tmp*np.array(self.alpha_list).reshape(-1,1)
        return np.sign(np.sum(tmp,axis=0))
```

预测结果如下，可以看到自写的 AdaBoost 分类器在稳定性上略逊一筹：

```
clf = [AdaBoost(n_estimators=i+1) for i in range(10)]
for c in clf:
    c.fit(X,Y)
    print('基分类器个数: ',clf.index(c)+1,'准确率: ',\
      np.sum(c.predict(X)==Y.reshape(-1))/len(Y))
```

```
## 基分类器个数： 1 准确率： 0.7
## 基分类器个数： 2 准确率： 0.4
## 基分类器个数： 3 准确率： 1.0
## 基分类器个数： 4 准确率： 0.7
## 基分类器个数： 5 准确率： 0.7
## 基分类器个数： 6 准确率： 1.0
## 基分类器个数： 7 准确率： 1.0
## 基分类器个数： 8 准确率： 1.0
## 基分类器个数： 9 准确率： 1.0
## 基分类器个数： 10 准确率： 1.0
```

梯度提升树

1 梯度提升树

1.1 前向分步算法

考虑加法模型：

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

其中 $b(x; \gamma_m)$ 为基函数，γ_m 为基函数的参数，β_m 为基函数的系数。

在给定训练数据和损失函数 $L(y, f(x))$ 时，学习加法模型 $f(x)$ 成为最小化以下经验风险极小化问题：

$$\min_{\beta_m, \gamma_m} \sum_{i=1}^{N} L(y_i, \sum_{m=1}^{M} \beta_m b(x_i; \gamma_m))$$

这个问题有 $2M$ 个参数，直接优化较为复杂。前向分步算法求解这一优化问题的思想是：从前向后不断学习，每一步只学习一个基函数及其系数，简化其复杂度。即每步只需优化以下函数：

$$\min_{\beta, \gamma} \sum_{i=1}^{N} L(y_i, \beta b(x_i; \gamma))$$

更一般地，参数 β_m, γ_m 可以如下求得：

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

并且由求出的 β_m, γ_m 更新 $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$。

这样，前向分步算法将同时求解 $m = 1, 2, ..., M$ 的所有参数 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。事实上，AdaBoost 就是一种前向分步算法。

1.2 提升树

以决策树为基函数的提升方法称为提升树。提升树可以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 为决策树的参数，M 为树的个数。使用前向分步算法，以 $f_0(x) = 0$ 为初始提升树，第 m 步的模型是：

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

当前已求得的模型为 $f_{m-1}(x)$，Θ_m 可由经验风险最小化确定：

$$\hat{\Theta}_m = arg \min_{\Theta_m} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + T(x_i; \Theta_m))$$

1.3 梯度提升树

以回归提升树为例，其使用平方误差损失函数 $L(y, f(x)) = (y - f(x))^2$。其损失变为：

$$\begin{aligned} L(y, f_{m-1}(x) + T(x; \Theta_m)) &= [y - f_{m-1}(x) - T(x; \Theta_m)]^2 \\ &:= [r - T(x; \Theta_m)]^2 \end{aligned}$$

这里 $r = y - f_{m-1}(x)$ 是当前模型的拟合残差。因此回归提升树只需简单拟合当前模型的残差。回归提升树算法如下：

输入：训练数据集 $T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}, x_i \in R^n, y_i \in R$ 和树的棵树 M。

1. 初始化 $f_0(x) = 0$
2. 对 $m = 1, 2, ..., M$

- 计算残差 $r_{mi} = y_i - f_{m-1}(x_i), \quad i = 1, 2, ..., N$
- 对残差 r_{mi} 学习一个回归树 $T(x; \Theta_m)$
- 更新 $f_m(x) = f_{m-1}(x) + T(x; \Theta_m)$

3. 得到回归问题的提升树 $f_M(x) = \sum_{m=1}^{M} T(x; \Theta_m)$

输出：提升树 $f_M(x)$。

回归提升树是一种特殊的梯度提升树。梯度提升树的主要思想是**不断地对数据的残差进行拟合**。但是很多时候，一般的损失函数的优化较为困难。我们可以使用损失函数的负梯度在当前模型的值来近似计算残差：

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

并对 r_{mi} 继续拟合一个决策树。回归提升树是梯度提升树的一种特例，因为：

$$-\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}|_{f_{m-1(x)}} = -\frac{\partial [(y_i - f(x_i))^2]}{\partial f(x_i)}|_{f_{m-1(x)}} = 2(y_i - f_{m-1}(x_i))$$

因此，$r = y - f_{m-1}(x)$。

2 代码实现

考虑以下数据：

x	y
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

2.1 sklearn 实现

准备数据：

```
import numpy as np
X = np.array([i+1 for i in range(10)]).reshape(-1,1)
Y = np.array([5.56,5.70,5.91,6.40,6.80,7.05,8.90,8.70,9.00,9.05]).reshape(-1,1)
```

梯度提升回归树的接口位于 sklearn.ensemble.GradientBoostingRegressor, 其文档可见**此处**。其中较为重要的参数有：

- n_estimators：基函数个数
- loss,criterion：优化函数和决策树切割标准
- 部分决策树的参数

```
from sklearn.ensemble import GradientBoostingRegressor
clf = [GradientBoostingRegressor(n_estimators=i+1,max_depth=1) for i in range(8)]
for c in clf:
    c.fit(X,Y.reshape(-1))
```

```
for c in clf:
    print('基分类器个数: ',clf.index(c)+1,'MSE: ',round(np.sum((c.predict(X)-Y.reshape(-1))**2),3))
```

```
## 基分类器个数:  1 MSE:  15.849
## 基分类器个数:  2 MSE:  13.205
## 基分类器个数:  3 MSE:  11.062
## 基分类器个数:  4 MSE:  9.327
## 基分类器个数:  5 MSE:  7.906
## 基分类器个数:  6 MSE:  6.722
## 基分类器个数:  7 MSE:  5.706
## 基分类器个数:  8 MSE:  4.874
```

2.2 回归提升树

使用回归提升树时，需要使用到 CART 回归树，需要将其导入。

```
import numpy as np
from CART import RegressionTree

class BoostingTree:

    def __init__(self,n_estimators=10,max_depth=1):
        self.n_estimators = n_estimators
        self.max_depth = max_depth

    def fit(self,X,Y):
        self.clf_list = []
        for m in range(self.n_estimators):
            clf = RegressionTree(max_depth = self.max_depth)
            clf.fit(X,Y)
            self.clf_list.append(clf)
```

```
            Y = Y-clf.predict(X)

    def predict(self,new_X):
        return np.sum(np.array([clf.predict(new_X) for clf in self.clf_list]),axis=0)
```

结果如下：

```
clf = [BoostingTree(n_estimators=i+1,max_depth=1) for i in range(6)]
for c in clf:
    c.fit(X,Y)

for c in clf:
    print('基分类器个数：',clf.index(c)+1,'MSE：',round(np.sum((c.predict(X)-Y)**2),3))
```

```
## 基分类器个数： 1 MSE： 1.93
## 基分类器个数： 2 MSE： 0.801
## 基分类器个数： 3 MSE： 0.478
## 基分类器个数： 4 MSE： 0.306
## 基分类器个数： 5 MSE： 0.229
## 基分类器个数： 6 MSE： 0.172
```

随机森林

1 随机森林

1.1 Bagging

AdaBoost 和梯度提升树都属于提升算法，即使用一系列的弱分类器**先后**进行训练，并将其进行组合，构成一个强分类器。在这些基分类器的训练过程中，后一个分类器需要前一个分类器的结果才可继续进行训练。

提升方法是**集成**方法的一种，集成方法还包括了 Bagging，Bagging 则是每次从数据集中采样出一部分的数据，在这些数据上训练出一个基分类器，最后将所有基分类器组合以得到一个强分类器。

其中需要注意的是：

1. 为了保证基分类器具有一定的差异，Bagging 使用了随机采样的方式，只从数据集中选取一部分数据进行基分类器的训练
2. 为了保证基分类器具有一定的学习能力，Bagging 对数据的采样方式为有交叠的采样子集，往往采用自助法
3. 与 Boosting 方法需要从前至后训练不同，Bagging 方法可以同时对基分类器进行训练，达到并行以加快训练速度
4. 使用自助法进行采样时，会有一部分样本未被抽取出，该部分样本可作为验证数据，进行超参数调试、决策树剪枝等工作

1.2 随机森林

随机森林是以决策树为基分类器构建 Bagging 集成的基础上，进一步在决策树训练过程中引入了随机属性选择。即对于决策树的每个节点，其先从该节点的属性集合中随机选择一个包含 k 个属性的子集，并从这个子集选取一个最优属性进行划分。

通过样本的随机采样和属性的随机选择，随机森林中的基分类器的多样性大大增加，其泛化能力因此也有提升。随机森林的算法如下：

输入：数据集 $T = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$，其中，$x_i \in R^n, y_i \in Y, i = 1, 2, ..., N$，基分类器个数 M，样本采样率 α 和属性采样率 $beta$。

1. 对 $m = 1, 2, ..., M$ 并行进行操作：

- 对 T 中的数据按概率 α 进行采样得到当前轮次的训练数据集 T_m
- 在数据集 T_m 上训练一个决策树，注意在对决策树节点进行划分时，需要先按 β 随机选择属性，再在剩下的属性选择最优的分割点
- 得到当前轮次的决策树模型 $f_m(x)$

2. 组合基分类器，得到最终分类器

$$F(x) = arg \max_{y \in Y} \sum_{m=1}^{M} I(f_m(x) = y)$$

输出：分类器 $F(x)$。

2 代码实现

考虑的数据集为 glass 数据集，具体可见 csv 文件。

2.1 sklearn 实现

准备数据：

```
import pandas as pd
import numpy as np
data = pd.read_csv('glass.csv')
X = data.iloc[:,:-1].values
Y = data.iloc[:,-1].values
```

随机森林的接口位于 sklearn.ensemble.RandomForestClassifier, 其文档可见**此处**。其中较为重要的参数有：

- n_estimators：基分类器的个数
- max_samples：样本采样比例
- max_features：属性采样比例
- 其余部分关于决策树的参数

```
from sklearn.ensemble import RandomForestClassifier
clf = [RandomForestClassifier(n_estimators=i,max_samples=0.8,max_features=0.8)\
       for i in range(5,101,5)]
for c in clf:
    c.fit(X,Y)
```

```
for c in clf:
    print('基分类器个数：',c.n_estimators,'准确率：',round(np.sum(c.predict(X)==Y)/len(Y),3))
```

```
## 基分类器个数： 5 准确率： 0.925
## 基分类器个数： 10 准确率： 0.958
## 基分类器个数： 15 准确率： 0.981
## 基分类器个数： 20 准确率： 0.977
## 基分类器个数： 25 准确率： 0.995
## 基分类器个数： 30 准确率： 0.995
## 基分类器个数： 35 准确率： 0.991
## 基分类器个数： 40 准确率： 1.0
## 基分类器个数： 45 准确率： 0.991
## 基分类器个数： 50 准确率： 0.995
## 基分类器个数： 55 准确率： 0.995
## 基分类器个数： 60 准确率： 1.0
## 基分类器个数： 65 准确率： 0.995
## 基分类器个数： 70 准确率： 1.0
## 基分类器个数： 75 准确率： 1.0
## 基分类器个数： 80 准确率： 1.0
## 基分类器个数： 85 准确率： 1.0
## 基分类器个数： 90 准确率： 1.0
## 基分类器个数： 95 准确率： 0.995
## 基分类器个数： 100 准确率： 1.0
```

2.2 随机森林

随机森林需要用到决策树，且该决策树在分裂节点时，只需计算部分属性的数值。从 utils 文件中导入该特定决策树即可。

```
class RandomForest:

    def __init__(self,n_estimators=100,subsample=1,colsample=1,\
                 max_depth=float('inf'),min_samples_leaf=1):
        self.n_estimators = n_estimators
        self.subsample = subsample
        self.colsample = colsample
        self.max_depth = max_depth
        self.min_samples_leaf = min_samples_leaf
```

```
    def fit(self,X,Y):
        self.clf_list = []
        for m in range(self.n_estimators):
            idx = np.random.permutation([i for i in range(len(Y))])
            idx = list(idx)[:round(self.subsample*len(Y))]
            X_m, Y_m = X[idx,:],Y[idx]
            clf = ClassificationTree(max_depth=self.max_depth,min_samples_leaf=self.min_samples_le
            clf.fit(X_m,Y_m,col_num=round(X.shape[1]*self.colsample))
            self.clf_list.append(clf)

    def find_most_frequent(self,x):
        return np.bincount(list(x)).argmax()

    def predict(self,new_X):
        out = np.concatenate([clf.predict(new_X) for clf in self.clf_list],axis=1)
        return np.apply_along_axis(self.find_most_frequent,axis=1,arr=out)
```

结果如下：

```
from utils import ClassificationTree
clf = [RandomForest(n_estimators=i) for i in range(1,10)]
for c in clf:
    c.fit(X,Y)
    print('基分类器个数: ',c.n_estimators,'准确率: ',round(np.sum(c.predict(X)==Y.reshape(-1))/len(Y
```

```
## 基分类器个数:  1 准确率:  0.972
## 基分类器个数:  2 准确率:  0.888
## 基分类器个数:  3 准确率:  0.972
## 基分类器个数:  4 准确率:  0.981
## 基分类器个数:  5 准确率:  1.0
## 基分类器个数:  6 准确率:  0.981
## 基分类器个数:  7 准确率:  0.995
## 基分类器个数:  8 准确率:  0.995
## 基分类器个数:  9 准确率:  0.995
```

EM 算法

1 EM 算法

1.1 算法的导出

概率模型可以通过观测变量建立极大似然函数，并通过最大化极大似然函数得出参数的估计。不过有时候，模型不仅有观测变量，还含有**隐变量**或**潜在变量**。隐变量无法观测但会影响数据的分布形式。

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, Z 连在一起时称为完全数据，观测数据 Y 称为不完全数据。对于含有隐变量的概率模型，目标是**极大化不完全数据 Y 关于参数 θ 的对数似然函数**，即极大化：

$$\begin{aligned} L(\theta) :&= \log P(Y|\theta) = \log \sum_Z P(Y, Z|\theta) \\ &= \log(\sum_Z P(Y|Z, \theta)P(Z|\theta)) \end{aligned}$$

我们希望极大化 $L(\theta)$。假设在第 i 次迭代后 θ 的估计值是 θ^i，我们希望找到新的估计值 θ 使得 $L(\theta) > L(\theta^i)$，并逐步达到最大值。为此，考虑：

$$L(\theta) - L(\theta^i) = \log(\sum_Z P(Y|Z, \theta)P(Z|\theta)) - \log P(Y|\theta^i)$$

利用 Jansen 不等式，有：

$$\begin{aligned} L(\theta) - L(\theta^i) &= \log(\sum_Z P(Z|Y, \theta^i)\frac{P(Y|Z, \theta)P(Z|\theta)}{P(Z|Y, \theta^i)}) - \log P(Y|\theta^i) \\ &\geq \sum_Z P(Z|Y, \theta^i) \log \frac{P(Y|Z, \theta)P(Z|\theta)}{P(Z|Y, \theta^i)} - \sum_Z P(Z|Y, \theta^i) \log P(Y|\theta^i) \\ &= \sum_Z P(Z|Y, \theta^i) \log \frac{P(Y|Z, \theta)P(Z|\theta)}{P(Z|Y, \theta^i)P(Y|\theta^i)} \end{aligned}$$

令

$$B(\theta,\theta^i) := L(\theta^i) + \sum_Z P(Z|Y,\theta^i)\log\frac{P(Y|Z,\theta)P(Z|\theta)}{P(Z|Y,\theta^i)P(Y|\theta^i)}$$

则有 $L(\theta) \geq B(\theta,\theta^i)$。因此 $B(\theta,\theta^i)$ 是 $L(\theta)$ 的一个下界。且易知 $L(\theta^i) = B(\theta^i,\theta^i)$。因此任何使 $B(\theta,\theta^i)$ 增大的 θ，也可以使 $L(\theta)$ 变大。为了使 $L(\theta)$ 尽可能地大，选择 θ^{i+1} 使得 $B(\theta,\theta^i)$ 达到极大，即：

$$\theta^{i+1} = arg\max_\theta B(\theta,\theta^i)$$

略去 $B(\theta,\theta^i)$ 中与 θ 无关的常数项，则：

$$\begin{aligned}
\theta^{i+1} &= arg\max_\theta [L(\theta^i) + \sum_Z P(Z|Y,\theta^i)\log\frac{P(Y|Z,\theta)P(Z|\theta)}{P(Z|Y,\theta^i)P(Y|\theta^i)}] \\
&= arg\max_\theta [\sum_Z P(Z|Y,\theta^i)\log(P(Y|Z,\theta)P(Z|\theta))] \\
&= arg\max_\theta [\sum_Z P(Z|Y,\theta^i)\log P(Y,Z|\theta)] \\
&:= arg\max_\theta Q(\theta,\theta^i)
\end{aligned}$$

因此，对于不完全数据的参数估计问题，可以转化为如下重复的两步：

1. **E 步**：根据当前的 θ^i 计算 $Q(\theta,\theta^i)$：

$$\begin{aligned}
Q(\theta,\theta^i) &= E_Z[\log P(Y,Z|\theta)|Y,\theta^i] \\
&= \sum_Z \log P(Y,Z|\theta)P(Z|Y,\theta^i)
\end{aligned}$$

2. **M 步**：求 $\theta^i = arg\max_\theta Q(\theta,\theta^i)$

因此，该算法被称为 EM 算法。EM 算法步骤如下：

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

1. 选择参数的初始值 θ^0，开始迭代
2. E 步：计算 $Q(\theta,\theta^i)$
3. M 步：得到 $\theta^i = arg\max_\theta Q(\theta,\theta^i)$
4. 重复 2,3 直至收敛

输出：模型参数 θ。

注意：EM 算法对初值很敏感，可能会收敛到局部最优值，可尝试选取多个初值进行迭代。

1.2 收敛性分析

由于 $P(Y|\theta)=\frac{P(Y,Z|\theta)}{P(Z|Y,\theta)}$，因此

$$\log P(Y|\theta)=\log P(Y,Z|\theta)-\log P(Z|Y,\theta)$$

令

$$Q(\theta,\theta^i)=\sum_Z \log P(Y,Z|\theta)P(Z|Y,\theta^i)$$
$$H(\theta,\theta^i)=\sum_Z \log P(Z|Y,\theta)P(Z|Y,\theta^i)$$

因此，$\log P(Y|\theta)=Q(\theta,\theta^i)-H(\theta,\theta^i)$，所以：

$$\begin{aligned}L(\theta^{i+1})-L(\theta^i)&=\log P(Y|\theta^{i+1})-\log P(Y|\theta^i)\\&=[Q(\theta^{i+1},\theta^i)-Q(\theta^i,\theta^i)]-[H(\theta^{i+1},\theta^i)-H(\theta^i,\theta^i)]\end{aligned}$$

由于 θ^{i+1} 使得 $Q(\theta,\theta^i)$ 达到极大，因此 $Q(\theta^{i+1},\theta^i)-Q(\theta^i,\theta^i)\geq 0$。此外：

$$\begin{aligned}H(\theta^{i+1},\theta^i)-H(\theta^i,\theta^i)&=\sum_Z(\log\frac{P(Z|Y,\theta^{i+1})}{P(Z|Y,\theta^i)})P(Z|Y,\theta^i)\\&\leq\log(\sum_Z\frac{P(Z|Y,\theta^{i+1})}{P(Z|Y,\theta^i)}P(Z|Y,\theta^i))\\&\leq\log\sum_Z P(Z|Y,\theta^{i+1})=0\end{aligned}$$

因此 $L(\theta^{i+1})\geq L(\theta^i)$。当 $Q(\theta,\theta^{'})$ 和 $L(\theta)$ 满足一定条件时（一般情况下很容易达到），EM 算法可以收敛到 $L(\theta)$ 的稳定点（因此不能保证收敛到极大值点）。

1.3 高斯混合模型

EM 算法一个重要应用场景是高斯混合模型。

定义：高斯混合模型是具有如下形式的概率分布模型：

$$P(y|\theta)=\sum_{k=1}^K\alpha_k\phi(y|\theta_k)$$

其中，$\alpha_k\geq 0$ 是系数，$\sum_{k=1}^K\alpha_k=1$，$\phi(y|\theta_k)=\phi(y|(\mu_k,\sigma_k^2))$ 是高斯分布密度

$$\phi(y|\theta_k) = \frac{1}{\sqrt{2\pi}\sigma_k}\exp(-\frac{(y-\mu_k)^2}{2\sigma_k^2})$$

假设观测数据 $y_1, y_2, ..., y_N$ 有高斯混合模型 $P(y|\theta) = \sum_{k=1}^{K}\alpha_k\phi(y|\theta_k)$ 生成，则可使用 EM 算法估计参数 $\theta = (\alpha_1, \alpha_2, ..., \alpha_K; \theta_1, \theta_2, ..., \theta_K)$ 的值。

该模型的隐变量 γ_{jk} 可定义为：

$$\gamma_{jk} = \begin{cases} 1 & , y_j \in model_k \\ 0 & , else \end{cases}$$

因此，对于完全数据 $(y_j, \gamma_{j1}, \gamma_{j2}, ..., \gamma_{jK}), j = 1, 2, ..., N$，完全数据的似然函数：

$$\begin{aligned} P(y, \gamma|\theta) &= \prod_{j=1}^{N} P(y_j, \gamma_{j1}, \gamma_{j2}, ..., \gamma_{jK}|\theta) \\ &= \prod_{k=1}^{K}\prod_{j=1}^{N}[\alpha_k\phi(y_j|\theta_k)]^{\gamma_{jk}} \\ &= \prod_{k=1}^{K}\alpha_k^{n_k}\prod_{j=1}^{N}[\phi(y_j|\theta_k)]^{\gamma_{jk}} \\ &= \prod_{k=1}^{K}\alpha_k^{n_k}\prod_{j=1}^{N}[\frac{1}{\sqrt{2\pi}\sigma_k}\exp(-\frac{(y_j-\mu_k)^2}{2\sigma_k^2})]^{\gamma_{jk}} \end{aligned}$$

其中，$n_k = \sum_{j=1}^{N}\gamma_{jk}, \sum_{k=1}^{K} = N$。因此，完全数据的对数似然函数是：

$$\log P(y, \gamma|\theta) = \sum_{k=1}^{K}\{n_k\log\alpha_k + \sum_{j=1}^{N}\gamma_{jk}[\log(\frac{1}{\sqrt{2\pi}}) - \log\sigma_k - \frac{1}{2\sigma_k^2}(y_j-\mu_k)^2]\}$$

所以，高斯混合模型的 **E 步**为确定 Q 函数：

$$\begin{aligned} Q(\theta, \theta^i) &= E_Z[\log P(Y, Z|\theta)|Y, \theta^i] \\ &= E_{\gamma_{jk}}[\sum_{k=1}^{K}\{n_k\log\alpha_k + \sum_{j=1}^{N}\gamma_{jk}[\log(\frac{1}{\sqrt{2\pi}}) - \log\sigma_k - \frac{1}{2\sigma_k^2}(y_j-\mu_k)^2]\}] \\ &= \sum_{k=1}^{K}\{\sum_{j=1}^{N}(E\gamma_{jk})\log\alpha_k + \sum_{j=1}^{N}(E\gamma_{jk})[\log(\frac{1}{\sqrt{2\pi}}) - \log\sigma_k - \frac{1}{2\sigma_k^2}(y_j-\mu_k)^2]\} \end{aligned}$$

其中，

$$\begin{aligned}\hat{\gamma}_{jk} &= E(\gamma_{jk}|y,\theta) = P(\gamma_{jk}=1|y,\theta) \\ &= \frac{P(\gamma_{jk}=1, y_j|\theta)}{P(y_j|\theta)} \\ &= \frac{P(\gamma_{jk}=1, y_j|\theta)}{\sum_{k=1}^{K} P(\gamma_{jk}=1, y_j|\theta)} \\ &= \frac{P(y_j|\gamma_{jk}=1,\theta)P(\gamma_{jk}=1|\theta)}{\sum_{k=1}^{K} P(y_j|\gamma_{jk}=1,\theta)P(\gamma_{jk}=1|\theta)} \\ &= \frac{\alpha_k \phi(y_j|\theta)}{\sum_{k=1}^{K} \alpha_k \phi(y_j|\theta)}, \quad j=1,2,...,N; k=1,2,...,K\end{aligned}$$

代入即有：

$$Q(\theta,\theta^i) = \sum_{k=1}^{K}\{\sum_{j=1}^{N} \hat{\gamma}_{jk} \log \alpha_k + \sum_{j=1}^{N} \hat{\gamma}_{jk}[\log(\frac{1}{\sqrt{2\pi}}) - \log \sigma_k - \frac{1}{2\sigma_k^2}(y_j-\mu_k)^2]\}$$

高斯混合模型的 **M 步**即为最大化函数 $Q(\theta,\theta^i)$。添加上约束条件 $\sum_{k=1}^{K} \alpha_k = 1$，使用拉格朗日乘数法可以得到以下估计：

$$\begin{aligned}\hat{\mu}_k &= \frac{\sum_{j=1}^{N} \hat{\gamma}_{jk} y_j}{\sum_{j=1}^{N} \hat{\gamma}_{jk}} && ,k=1,2,...,K \\ \hat{\sigma}_k^2 &= \frac{\sum_{j=1}^{N} \hat{\gamma}_{jk}(y_j-\mu_k)^2}{\sum_{j=1}^{N} \hat{\gamma}_{jk}} && ,k=1,2,...,K \\ \hat{\alpha}_k &= \frac{n_k}{N} = \frac{\sum_{j=1}^{N} \hat{\gamma}_{jk}}{N} && ,k=1,2,...,K\end{aligned}$$

重复 E 步和 M 步，直到对数似然函数值不再有明显的变化为止。高斯混合模型的 EM 算法步骤如下：

输入：观测数据 $y_1, y_2, ..., y_N$ 和分模型个数 K。

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

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

3. M 步：计算新一轮迭代的模型参数

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

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

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

4. 重复步骤 2 和 3，直到收敛

输出：高斯混合模型的参数。

2 代码实现

本次使用的数据为-67,-48,6,8,14,16,23,24,28,29,41,49,56,60,75。分模型个数为 2。

2.1 sklearn 实现

准备数据：

```
import numpy as np
X = np.array([-67,-48,6,8,14,16,23,24,28,29,41,49,56,60,75]).reshape(-1,1)
```

高斯混合模型的接口位于 sklearn.mixture.GaussianMixture, 其文档可见**此处**。其中较为重要的参数有：

- n_components：分模型个数
- max_iter：最大迭代次数
- weights_init,means_init：自主设置初始化参数

```
from sklearn.mixture import GaussianMixture
clf = GaussianMixture(n_components=2)
clf.fit(X)
```

```
print('权重：',clf.weights_,'\n平均值：\n',clf.means_,'\n方差：\n',clf.covariances_[:,:,0])
```

```
## 权重： [0.13317238 0.86682762]
## 平均值：
##  [[-57.51107027]
##  [ 32.98489643]]
## 方差：
```

```
##  [[ 90.24987882]
##  [429.45764867]]
```

2.2 高斯混合模型

此处只实现一维情况下的高斯混合模型。同时模型的初值是随机从原始数据中选取两个值得到的。

```
import math
import random

class GaussianMixture:

    def __init__(self,n_components,esp=1e-3):
        self.n_components = n_components
        self.esp = esp

    def gaussian(self,x,mu,sigma):
        return 1/math.sqrt(2*math.pi*sigma)*math.exp(-(x-mu)**2/(2*sigma))


    def fit(self,X):
        self.alpha = np.array([1/self.n_components for _ in range(self.n_components)])
        self.mean = np.array(random.sample(list(X),2)).reshape(-1,1)
        self.sigma = np.array([np.var(X).reshape(1,1)/self.n_components for _ in range(self.n_comp
        err = self.esp+1

        while err>self.esp:
            mean = self.mean
            sigma = self.sigma
            alpha = self.alpha

            ## compute gamma
            gamma = []
            for k in range(self.n_components):
                for j in range(len(X)):
                    gamma.append(self.gaussian(X[j,:],self.mean[k],self.sigma[k]))
            gamma = np.array(gamma).reshape(self.n_components,-1)
            gamma = gamma/np.sum(gamma,axis=0)
```

```
            ## compute mu,sigma,alpha
            self.mean = np.array([np.sum(gamma[k,:].reshape(-1,1)*X)/np.sum(gamma[k,:]) for k in ra
            self.sigma = np.array([(np.sum(gamma[k,:].reshape(-1,1)*(X-self.mean[k])**2)\
                                    /np.sum(gamma[k,:]))\
                                    for k in range(self.n_components)])
            self.alpha = (np.sum(gamma,axis=1)/X.shape[0]).reshape(-1)

            err = np.mean(np.abs(mean-self.mean))+np.mean(np.abs(sigma-self.sigma))+\
                    np.mean(np.abs(alpha-self.alpha))
```

在不同的种子下，EM 算法输出的结果不同：

```
random.seed(15)
clf = GaussianMixture(2)
clf.fit(X)
print('权重: ',clf.alpha,'\n平均值: \n',clf.mean,'\n方差: \n',clf.sigma)
```

```
## 权重:  [0.86669097 0.13330903]
## 平均值:
##  [ 32.99771897 -57.50167052]
## 方差:
##  [428.48138525  90.24999966]
```

```
random.seed(25)
clf = GaussianMixture(2)
clf.fit(X)
print('权重: ',clf.alpha,'\n平均值: \n',clf.mean,'\n方差: \n',clf.sigma)
```

```
## 权重:  [0.4532828 0.5467172]
## 平均值:
##  [ 6.61029534 32.80855304]
## 方差:
##  [2127.19989767  357.31428709]
```

隐马尔可夫模型

1 隐马尔可夫模型

1.1 模型定义

定义： 隐马尔可夫模型是关于时序的概率模型，描述一个由**隐藏的马尔可夫链**生成**不可观测的状态随机序列**，再有各个状态生成一个**观测随机序列**的过程。

其中，隐藏的马尔可夫链随机生成*状态序列*；每个状态生成一个观测，由此产生的观测随机序列，称为*观测序列*。

隐马尔可夫模型可由**初始概率分布、状态转移概率分布和观测概率分布**确定。可按如下数学形式进行严格定义：

设 $Q=\{q_1,q_2,...,q_N\}$ 是所有状态的集合，$V=\{v_1,v_2,...,v_M\}$ 是所有观测的集合。$I=(i_1,i_2,...,i_T)$ 是转态序列，$O=(o_1,o_2,...,o_T)$ 是观测序列。

*状态的转移矩阵*为 $A=[a_{ij}]_{N\times N}$，其中 $a_{ij}=P(i_{t+1}=q_j|i_t=q_i),i=1,2,...,N,j=1,2,...,N$ 表示在时刻 t 下不可观测的状态序列从 q_i 转移到 q_j 的概率。

*状态观测转移的观测概率矩阵*为 $B=[b_j(k)]_{N\times M},j=1,2,...,N,k=1,2,...,M$，其中 $b_j(k)=P(o_t=v_k|i_t=q_j)$ 表示在时刻 t 下不可观测的状态 q_j 转化为可观测的观测 v_k 的概率。

*状态的初始概率向量*为 $\pi=(\pi_i)$，其中 $\pi_i=P(i_1=q_i),i=1,2,...,N$ 表示 t_1 时刻下处于状态 q_i 的概率。

隐马尔可夫模型由初始状态概率向量 π，状态转移概率矩阵 A 和观测概率矩阵 B 决定。其中状态序列由 π,A 决定，观测序列由 B 决定。因此，隐马尔可夫模型可表示为三元组：$\lambda=(A,B,\pi)$。

隐马尔可夫模型具有以下两个基本假设：

1. 马尔可夫假设：$P(i_t|i_{t-1},o_{t-1},...,i_1,o_1)=P(i_t|i_{t-1})$
2. 观测独立性假设：$P(o_t|o_T,i_T,o_{T-1},i_{T-1},...,o_1,i_1)=P(o_t|i_t)$

隐马尔可夫模型具有以下三个基本问题：

1. **概率计算问题**：给定模型 $\lambda=(A,B,\pi)$ 和观测序列 $O=(o_1,o_2,...,o_T)$，计算在 λ 下观测序列 O 出现的概率 $P(O|\lambda)$

2. **学习问题**：已知观测序列 $O=(o_1,o_2,...,o_T)$，估计模型 $\lambda=(A,B,\pi)$ 的参数
3. **预测问题**：也称为解码问题。已知模型 $\lambda=(A,B,\pi)$ 和观测序列 $O=(o_1,o_2,...,o_T)$，求最有可能的状态序列 $I=(i_1,i_2,...,i_T)$

1.2 概率计算算法

1.2.1 直接计算法

对于给定的状态序列 $I=(i_1,i_2,...,i_T)$，其出现概率为

$$P(I|\lambda)=\pi_{i_1}a_{i_1i_2}a_{i_2i_3}...a_{i_{T-1}i_T}$$

对于固定的状态序列 $I=(i_1,i_2,...,i_T)$ 和观测序列 $O=(o_1,o_2,...,o_T)$ 出现的概率为

$$P(O|I)=b_{i_1}(o_1)b_{i_2}(o_2)...b_{i_T}(o_T)$$

因此，O,I 的联合概率为

$$P(O,I|\lambda)=P(O|I,\lambda)P(I|\lambda)=\pi_{i_1}b_{i_1}(o_1)a_{i_1i_2}b_{i_2}(o_2)...a_{i_{T-1}i_T}b_{i_T}(o_T)$$

对所有 I 求和即可得到观测序列 O 出现的概率：

$$\begin{aligned}P(O|\lambda)&=\sum_I P(O,I|\lambda)\\&=\sum_{i_1,i_2,...,i_T}\pi_{i_1}b_{i_1}(o_1)a_{i_1i_2}b_{i_2}(o_2)...a_{i_{T-1}i_T}b_{i_T}(o_T)\end{aligned}$$

此计算方法的计算复杂度为 $O(TN^T)$。因此在实际操作中计算量过大，并不可行，需要使用前向-后向算法减少计算量。

1.2.2 前向算法

定义：给定隐马尔可夫模型 λ，定义到时刻 t 部分观测序列为 $o_1,o_2,...,o_t$ 且状态为 q_i 的概率为**前向概率**，记为：

$$\alpha_t(i)=P(o_1,o_2,...,o_t,i_t=q_i|\lambda)$$

因此，可以获得以下递推式：

$$\begin{aligned}\alpha_{t+1}(i) &= P(o_1, o_2, ..., o_{t+1}, i_{t+1} = q_i|\lambda) \\ &= \sum_{j=1}^{N} P(o_1, o_2, ..., o_{t+1}, i_{t+1} = q_i, i_t = q_j|\lambda) \\ &= \sum_{j=1}^{N} P(o_1, o_2, ..., o_t, i_t = q_j|\lambda) P(i_{t+1} = q_i|i_t = q_j, \lambda) P(o_{t+1}|i_{t+1} = q_i\lambda) \\ &= [\sum_{j=1}^{N} \alpha_t(j) a_{ji}] b_i(o_{t+1})\end{aligned}$$

因此，可按如下步骤计算前向概率 $\alpha_t(i)$：

1. 初始化 $\alpha_1(i) = \pi_i b_i(o_1), i = 1, 2, ..., N$
2. 对 $t = 1, 2, ..., T-1$ 递推：

$$\alpha_{t+1}(i) = [\sum_{j=1}^{N} \alpha_t(j) a_{ji}] b_i(o_{t+1}), i = 1, 2, ..., N$$

3. 计算 $P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$

前向算法的计算复杂度是 $O(TN^2)$，其复杂度大幅度减少的原因是每一次计算直接引用了前一次计算的结果。

1.2.3 后向算法

定义：给定隐马尔可夫模型 λ，定义在时刻 t 状态为 q_i 的条件下，从 $t+1$ 到 T 的部分观测序列 $o_{t+1}, o_{t+2}, ..., o_T$ 的概率为**后向概率**，记为：

$$\beta_t(i) = P(o_{t+1}, o_{t+2}, ..., o_T|i_t = q_i, \lambda)$$

同样，对 $\beta_t(i)$ 也可获得如下递推式：

$$\begin{aligned}\beta_t(i) &= P(o_{t+1}, o_{t+2}, ..., o_T|i_t = q_i, \lambda) \\ &= \sum_{j=1}^{N} P(o_{t+1}, o_{t+2}, ..., o_T, i_{t+1} = q_j|i_t = q_i, \lambda) \\ &= \sum_{j=1}^{N} P(o_{t+2}, ..., o_T, |i_{t+1} = q_j, \lambda) P(i_{t+1} = q_j|i_t = q_i, \lambda) P(o_{t+1}|i_{t+1} = q_j, \lambda) \\ &= \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)\end{aligned}$$

因此，可按如下步骤计算后向概率 $\beta_t(i)$：

1. 初始化 $\beta_T(i)=1, i=1,2,...,N$
2. 对 $t=T-1,T-2,...,1$ 递推：

$$\beta_t(i)=\sum_{j=1}^{N}a_{ij}b_j(o_{t+1})\beta_{t+1}(j), i=1,2,...,N$$

3. 计算 $P(O|\lambda)=\sum_{i=1}^{N}\pi_i b_i(o_1)\beta_1(i)$

后向算法的时间复杂度同样为 $O(TN^2)$。

利用前向算法和后向算法，还可以得到关于单个状态和两个状态的概率计算公式

1. 给定模型 λ 和观测 O，模型在时刻 t 处于状态 q_i 的概率定义为 $\gamma_t(i)=P(i_t=q_i|O,\lambda)$。可以得到：

$$\gamma_t(i)=P(i_t=q_i|O,\lambda)=\frac{P(i_t=q_i,O|\lambda)}{P(O|\lambda)}=\frac{P(i_t=q_i,O|\lambda)}{\sum_{j=1}^{N}P(i_t=q_j,O|\lambda)}=\frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{N}\alpha_t(j)\beta_t(j)}$$

2. 给定模型 λ 和观测 O，模型在时刻 t 处于状态 q_i 且在时刻 $t+1$ 处于状态 q_j 的概率定义为 $\xi_t(i,j)=P(i_t=q_i,i_{t+1}=q_j|O,\lambda)$。可以得到：

$$\begin{aligned}\xi_t(i,j)=P(i_t=q_i,i_{t+1}=q_j|O,\lambda)&=\frac{P(i_t=q_i,i_{t+1}=q_j,O|\lambda)}{P(O|\lambda)}\\&=\frac{P(i_t=q_i,i_{t+1}=q_j,O|\lambda)}{\sum_{i=1}^{N}\sum_{j=1}^{N}P(i_t=q_i,i_{t+1}=q_j,O|\lambda)}\\&=\frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}\end{aligned}$$

1.3 学习算法

1.3.1 监督学习算法

当状态序列和观测序列均是已知时，通过序列 $\{(O_1,I_1),(O_2,I_2),...,(O_S,I_S)\}$ 使用极大似然估计可以简单得到 π,A,B 中的参数。

$$\hat{a}_{ij}=\frac{A_{ij}}{\sum_{i=1}^{N}\sum_{j=1}^{N}A_{ij}},\hat{b}_j(k)=\frac{B_{jk}}{\sum_{k=1}^{M}B_{jk}},\hat{\pi}_i=\frac{\Pi_i}{\sum_{j=1}^{N}\Pi_j}$$

其中，A_{ij} 表示从状态 q_i 转移到状态 q_j 的频数，B_{jk} 表示从状态 b_j 变化为观测 o_k 的频数，Π_i 表示首个状态为 q_i 的频数。

1.3.2 Baum-Welch 算法

当只知道观测序列 $\{O_1,O_2,...,O_T\}$，而不知道状态序列 $\{I_1,I_2,...,I_T\}$ 时，需要使用 EM 算法对参数进行估计。此时，$\{O_1,O_2,...,O_T\}$ 是不完全数据，而 $\{(O_1,I_1),(O_2,I_2),...,(O_S,I_S)\}$ 是完全数据。

算法的 E 步是确定 Q 函数 $Q(\lambda, \bar{\lambda})$:

$$\begin{aligned} Q(\lambda, \bar{\lambda}) &= E_I[\log P(O, I|\lambda)|O, \bar{\lambda}] \\ &= \sum_I \log P(O, I|\lambda) P(I|O, \bar{\lambda}) \\ &= \sum_I \log P(O, I|\lambda) \frac{P(O, I|\bar{\lambda})}{P(O|\bar{\lambda})} \\ &\propto \sum_I \log P(O, I|\lambda) P(O, I|\bar{\lambda}) \end{aligned}$$

而 $P(O, I|\lambda) = \pi_{i_1} b_{i_1}(o_1) a_{i_1 i_2} b_{i_2}(o_2) ... a_{i_{T-1} i_T} b_{i_T}(o_T)$，因此：

$$Q(\lambda, \bar{\lambda}) = \sum_I \log \pi_{i_1} P(O, I|\bar{\lambda}) + \sum_I (\sum_{t=1}^{T-1} \log a_{i_t i_{t+1}}) P(O, I|\bar{\lambda}) + \sum_I (\sum_{t=1}^{T} \log b_{i_t}(o_t)) P(O, I|\bar{\lambda})$$

算法的 M 步需要极大化上述 Q 函数 $Q(\lambda, \bar{\lambda})$。注意到 λ 的参数 A, B, π 分别位于 Q 函数的三项之中，相互不影响，因此只要分别极大化每一项即可。注意参数 A, B, π 满足一定的约束条件：$\sum_{i=1}^{N} \pi_i = 1, \sum_{j=1}^{N} a_{ij} = 1, \sum_{k=1}^{M} b_j(k) = 1$，最大化时需要使用拉格朗日乘数法。得到的结果为：

$$\hat{\pi}_i = \frac{P(O, i_1 = i|\bar{\lambda})}{P(O|\bar{\lambda})} = \gamma_1(i)$$

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} P(O, i_t = i, i_{t+1} = j|\bar{\lambda})}{\sum_{t=1}^{T-1} P(O, i_t = i|\bar{\lambda})} = \frac{\sum_{t=1}^{T-1} \xi_t(i, j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

$$\hat{b}_j(k) = \frac{\sum_{t=1}^{T} P(O, i_t = j|\bar{\lambda}) I(o_t = v_k)}{\sum_{t=1}^{T} P(O, i_t = j|\bar{\lambda})} = \frac{\sum_{t=1}^{T} \gamma_t(j) I(o_t = v_k)}{\sum_{t=1}^{T} \gamma_t(j)}$$

EM 算法作用于隐马尔可夫模型的计算如上所示。其被称为 Baum-Welch 算法，具体步骤如下：

输入：观测数据 $O = (o_1, o_2, ..., o_T)$。

1. 初始化，选取 $a_{ij}^{(0)}, b_j(k)^{(0)}, \pi_i^{(0)}$，得到模型 $\lambda^{(0)} = (A^{(0)}, B^{(0)}, \pi^{(0)})$
2. 使用前向-后向算法计算 $\alpha_t(i), \beta_t(i)$
3. 计算 $\gamma_t(j), \xi_t(i, j)$，并更新参数得到 $\lambda^{(n)} = (A^{(n)}, B^{(n)}, \pi^{(n)})$：

$$\hat{\pi}_i = \gamma_1(i)$$

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i, j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

$$\hat{b}_j(k) = \frac{\sum_{t=1}^{T} \gamma_t(j) I(o_t = v_k)}{\sum_{t=1}^{T} \gamma_t(j)}$$

4. 重复 2 和 3 直至收敛

输出：隐马尔可夫模型参数 $\hat{A}, \hat{B}, \hat{\pi}$。

1.4 预测算法

1.4.1 近似算法

近似算法在每个时刻 t 分别选择该时刻最有可能出现的状态 i_t^*，从而得到一个状态序列 $I^* = (i_1^*, i_2^*, ..., i_T^*)$。由于 $\gamma_t(i) = \frac{P(i_t=q_i, O|\lambda)}{P(O|\lambda)}$，因此易于发现：

$$i_t^* = arg \max_{1 \le i \le N} \gamma_t(i)$$

近似算法比较简单，但没有从整体上考虑状态序列。

1.4.2 维特比算法

维特比算法使用动态规划求概率最大的路径。其思想是，要使从 i_1^* 到 i_T^* 的路径最优，那么对于任意的 $2 \le t \le T-1$，从 i_t^* 到 i_T^* 的路径都是最优的。因此维特比算法从 $t=1$ 开始，递推地向后计算各条路径的最大概率，直到 $t=T$。在计算完最大概率后，再从 i_T^* 开始，由后向前求解节点 $i_{T-1}^*, ..., i_1^*$，从而得出最优路径 $I^* = (i_1^*, i_2^*, ..., i_T^*)$。

在向前计算最大概率时，需要递推进行计算以提高效率。定义时刻 t 时状态为 i 的所有单个路径 $(i_1, i_2, ..., i_t)$ 中的概率最大值为：

$$\delta_t(i) = \max_{i_1, i_2, ..., i_{t-1}} P(i_t = i, i_{t-1}, ..., i_1, o_t, o_{t-1}, ..., o_1|\lambda), \quad i = 1, 2, ..., N$$

因此有

$$\begin{aligned} \delta_{t+1}(i) &= \max_{i_1, i_2, ..., i_t} P(i_{t+1} = i, i_t, ..., i_1, o_{t+1}, o_t, ..., o_1|\lambda) \\ &= \max_{1 \le j \le N} [\delta_t(j) a_{ji}] b_i(o_{t+1}), \quad i = 1, 2, .., N; t = 1, 2, ..., T-1 \end{aligned}$$

在反向寻找最优节点时，也可定义在时刻 t 时状态为 i 的所有单个路径 $(i_1, i_2, ..., i_{t-1}, i)$ 中概率最大的路径的第 $t-1$ 个节点为：

$$\Psi_t(i) = arg \max_{1 \le j \le N} [\delta_{t-1}(j) a_{ji}], \quad i = 1, 2, ..., N$$

使用函数 δ 和 Ψ，可以得到维特比算法，其步骤如下：

输入：模型 $\lambda = (A, B, \pi)$ 和观测 $O = \{o_1, o_2, ..., o_T\}$

1. 初始化

$$\delta_1(i) = \pi_i b_i(o_1), \quad i = 1, 2, ..., N$$

$$\Psi_1(i) = 0, \quad i = 1, 2, ..., N$$

2. 向前递推，对 $t = 1, 2, ..., T-1$，计算

$$\delta_{t+1}(i) = \max_{1 \le j \le N}[\delta_t(j)a_{ji}]b_i(o_{t+1}), \quad i = 1, 2, ..., N$$

$$\Psi_t(i) = arg \max_{1 \le j \le N}[\delta_{t-1}(j)a_{ji}], \quad i = 1, 2, ..., N$$

3. 找出最大概率 $P^* = \max_{1 \le i \le N} \delta_T(i)$ 并确定最终节点 $i_T^* = arg \max_{1 \le i \le N} \delta_T(i)$
4. 反向找出最优节点，对 $t = T-1, T-2, ..., 1$，计算

$$i_t^* = \Psi_{t+1}(i_{t+1}^*)$$

输出：最优路径 $I^* = (i_1^*, i_2^*, ..., i_T^*)$。

2 代码实现

本次考虑的隐马尔可夫模型参数如下：

$$\pi = (0.2, 0.4, 0.4)^T, A = \begin{bmatrix} 0.5 & 0.2 & 0.3 \\ 0.3 & 0.5 & 0.2 \\ 0.2 & 0.2 & 0.6 \end{bmatrix}, B = \begin{bmatrix} 0.5 & 0.5 \\ 0.4 & 0.6 \\ 0.7 & 0.3 \end{bmatrix}$$

从该隐马尔可夫模型中随机生成 50 个长度为 50 的观测序列作为此次的数据。

2.1 sklearn 实现

sklearn 中的隐马尔可夫模型的模块已被独立到 hmmlearn 包中。使用 MultinomialHMM 函数可以模拟或者拟合离散的隐马尔可夫模型，该函数的主要属性有：

- startprob_：初始状态概率向量
- transmat_：状态转移矩阵
- emissionprob_：观测概率矩阵

首先，根据隐马尔可夫模型的参数模拟出 50 个长度为 50 的观测序列：

```
import numpy as np
from hmmlearn.hmm import MultinomialHMM

np.random.seed(12345)
model = MultinomialHMM(n_components=3)
model.startprob_ = np.array([0.2, 0.4, 0.4])
model.transmat_ = np.array([[0.5, 0.2, 0.3],
                            [0.3, 0.5, 0.2],
                            [0.2, 0.3, 0.5]])
model.emissionprob_ = np.array([[0.5, 0.5],
                                [0.4, 0.6],
                                [0.7, 0.3]])
O = np.concatenate([model.sample(50)[0] for _ in range(50)],axis=0)
```

再创建一个新的类，对其进行拟合并输出结果：

```
remodel = MultinomialHMM(n_components=3)
remodel.fit(O,lengths=[50 for _ in range(50)])
```

```
print('初始状态概率向量:\n',remodel.startprob_,
      '\n状态转移矩阵:\n',remodel.transmat_,
      '\n观测概率矩阵:\n',remodel.emissionprob_)
```

```
## 初始状态概率向量:
##  [0.29689949 0.43891086 0.26418965]
## 状态转移矩阵:
##  [[0.34383868 0.29584149 0.36031983]
##  [0.34050106 0.3076312  0.35186774]
##  [0.34480482 0.29234502 0.36285016]]
## 观测概率矩阵:
##  [[0.60855692 0.39144308]
##  [0.22427814 0.77572186]
##  [0.71926203 0.28073797]]
```

2.2 预测算法

在类中实现以下几个函数：

- compute_alpha,compute_beta：分别为前向计算和后向计算
- fit：使用 Baum-Welch 算法进行学习，不过在样本量较小时学习效果一般

- approx,viterbi：分别使用近似算法和维特比算法进行状态序列的预测

```
class HiddenMarkovModel:

    def __init__(self,n_components=1,max_iter=100,\
                 pi=None,A=None,B=None):
        self.n_components = n_components
        self.max_iter = max_iter
        self.pi = pi
        self.A = A
        self.B = B

    def compute_alpha(self,X):
        T = len(X)
        alpha = np.zeros((T,self.n_components))
        for i in range(T):
            if i==0:
                alpha[i,:] = self.pi*self.B[:,X[i]].reshape(-1)
            else:
                alpha[i,:] = np.sum(self.A.T*alpha[i-1,:],1)*self.B[:,X[i]].reshape(-1)
        return alpha

    def compute_beta(self,X):
        T = len(X)
        beta = np.ones((T,self.n_components))
        for i in range(T-1,0,-1):
            beta[i-1,:] = np.sum(self.A*self.B[:,X[i]].reshape(1,-1)*beta[i,:],0)
        return beta

    def compute_gamma(self):
        return self.alpha*self.beta/np.sum(self.alpha*self.beta,1).reshape(-1,1)

    def compute_xi(self,X):
        T,N = self.alpha.shape
        xi = np.zeros((T-1,N,N))
        for t in range(0,T-1):
            for i in range(0,N):
                for j in range(0,N):
                    xi[t,i,j] = self.alpha[t,i]*self.A[i,j]*self.B[j,X[t+1]]*self.beta[t+1,j]
```

```
        return xi/np.sum(np.sum(xi,2),1).reshape(-1,1,1)

    def fit(self,X):
        N,K,T = self.n_components,len(np.unique(X)),len(X)

        self.pi = np.array([np.random.random() for _ in range(N)])
        self.pi = self.pi/np.sum(self.pi)

        self.A = np.array([np.random.random() for _ in range(N*N)]).reshape(N,N)
        self.A = self.A/np.sum(self.A,1).reshape(-1,1)

        self.B = np.array([np.random.random() for _ in range(N*K)]).reshape(N,K)
        self.B = self.B/np.sum(self.B,1).reshape(-1,1)



        self.alpha = np.zeros((T,N))
        self.beta = np.ones((T,N))
        self.xi = np.zeros((T-1,N,N))

        for _ in range(self.max_iter):
            self.alpha = self.compute_alpha(X)
            self.beta = self.compute_beta(X)
            self.gamma = self.compute_gamma()
            self.xi = self.compute_xi(X)

            self.A = np.sum(self.xi,axis=0).reshape(N,N)/np.sum(self.gamma[:-1,:],axis=0)
            self.B = np.concatenate([(np.sum((X==i)*self.gamma,0)/np.sum(self.gamma,0)).\
                    reshape(N,1) for i in range(K)],1)
            self.pi = self.gamma[0,:].reshape(-1)

    def approx(self,O):
        alpha = self.compute_alpha(O)
        beta = self.compute_beta(O)
        gamma = alpha*beta/np.sum(alpha*beta,1).reshape(-1,1)
        return np.argmax(gamma,1)

    def viterbi(self,O):
        T,N = len(O),self.n_components
```

```
        delta = np.zeros((T,N))
        psi = np.zeros((T,N))
        for t in range(T):
            if t==0:
                delta[t,:] = self.pi*self.B[:,O[t]].reshape(-1)
            else:
                delta[t,:] = np.max(self.A*delta[t-1,:],1)*self.B[:,O[t]].reshape(-1)
                psi[t,:] = np.argmax(self.A*delta[t-1,:],1)
        path = []
        max_prob = np.max(delta[-1,:])
        path.append(np.argmax(delta,1)[-1])
        for t in range(T-1,0,-1):
            path.append(int(psi[t,int(path[-1])]))
        return np.array(path[::-1])
```

模拟一个长度为 10 的序列，预测算法结果如下：

```
X, Z = model.sample(10)

clf = HiddenMarkovModel(n_components=3)
clf.pi = np.array([0.2, 0.4, 0.4])
clf.A = np.array([[0.5, 0.2, 0.3],
                  [0.3, 0.5, 0.2],
                  [0.2, 0.3, 0.5]])
clf.B = np.array([[0.5, 0.5],
                  [0.4, 0.6],
                  [0.7, 0.3]])

print('状态序列:',Z,'\n近似算法估计:',clf.approx(X),'\n维特比算法估计:',clf.viterbi(X))
```

```
## 状态序列: [1 1 2 1 0 1 2 1 1 1]
## 近似算法估计: [1 1 1 1 1 1 1 1 1 1]
## 维特比算法估计: [1 1 1 1 1 1 1 1 1 1]
```

聚类

1 聚类

1.1 相似度与距离

对数据进行聚类时，需要计算数据间的相似度或距离以决定数据是否被分到一类中。常用的相似度或距离的计算方法有：

1. 明科夫斯基距离（其中，$p = 1, 2$ 分别为曼哈顿距离和欧氏距离）：

$$d_{ij} = \left(\sum_{k=1}^{m} |x_{ki} - x_{kj}|^p \right)^{\frac{1}{p}}$$

2. 马氏距离。设 S 是 X 的协方差矩阵，则 x_i 和 x_j 的马氏距离为：

$$d_{ij} = [(x_i - x_j)^T S^{-1} (x_i - x_j)]^{\frac{1}{2}}$$

3. 相关系数：

$$r_{ij} = \frac{\sum_{k=1}^{m} (x_{ki} - \bar{x}_i)(x_{kj} - \bar{x}_j)}{\left[\sum_{k=1}^{m} (x_{ki} - \bar{x}_i)^2 \sum_{k=1}^{m} (x_{kj} - \bar{x}_j)^2 \right]^{\frac{1}{2}}}$$

4. 余弦夹角：

$$s_{ij} = \frac{\sum_{k=1}^{m} x_{ki} x_{kj}}{\left[\sum_{k=1}^{m} x_{ki}^2 \sum_{k=1}^{m} x_{kj}^2 \right]^{\frac{1}{2}}}$$

1.2 k 均值聚类

常用的聚类方法主要有两种：层次聚类和 k 均值聚类。层次聚类的时间复杂度和空间复杂度在数据量增大时都会急速增大，因此层次聚类不太适合大规模数据集的聚类。此处主要介绍 k 均值聚类。k 均值聚类是一种使用迭代的聚类方法。

k 均值聚类使用欧氏距离平方作为样本之间的距离度量，并且定义样本与其所属类的中心的距离的总和为损失函数，即

$$W(C) = \sum_{l=1}^{k} \sum_{C(i)=l} ||x_i - \bar{x}_l||^2$$

穷举所有可能情况会造成该优化问题变为一个 NP 困难问题。因此在此处只能使用迭代法进行求解。但是迭代法可能会使该损失函数落入一个局部极小值点。k 均值聚类的迭代方法重复以下步骤直至划分结果不再改变：

1. 对于给定的中心值 $(m_1, m_2, ..., m_k)$，求一个划分 C 使得目标函数最小化：

$$\min_{C} \sum_{l=1}^{k} \sum_{C(i)=l} ||x_i - m_l||^2$$

   易于发现，该问题的求解结果为：将每个样本分到与其最近的中心 m_l 的类 G_l 中。
2. 对于给定的划分 C，再求各个类的中心 $(m_1, m_2, ..., m_k)$ 使得目标函数最小化：

$$\min_{m_1, m_2, ..., m_k} \sum_{l=1}^{k} \sum_{C(i)=l} ||x_i - m_l||^2$$

   求解结果为：$m_l = \frac{1}{n_l} \sum_{G(i)=l} x_i, l = 1, 2, ..., k$。

因此，k 均值聚类的步骤是：

输入：n 个样本的基本 X 和类的个数 k。

1. 初始化：随机初始化或者随机选取 k 个样本点作为初始聚类中心 $m^0 = (m_1^0, m_2^0, ..., m_k^0)$
2. 对固定的类中心 $m^t = (m_1^t, m_2^t, ..., m_k^t)$，计算每个样本到类中心的距离，将每个样本点分到与其最近的中心的类中，构成聚类结果 C^t
3. 重新计算类中心，得到新的中心值 $m^{t+1} = (m_1^{t+1}, m_2^{t+1}, ..., m_k^{t+1})$
4. 重复 2,3，直到划分结果不再改变

输出：聚类结果 C^*。

2 代码实现

本次使用的数据为模拟数据，数据量为 300。数据可分为三组，每组各 100 个，中心分别为 $(0,0)^T, (2,2)^T, (4,4)^T$。

```
import numpy as np
np.random.seed(12345)
x1 = np.random.randn(100,2)
x2 = np.random.randn(100,2) + [2,2]
x3 = np.random.randn(100,2) + [4,4]
X = np.vstack((x1,x2,x3))
```

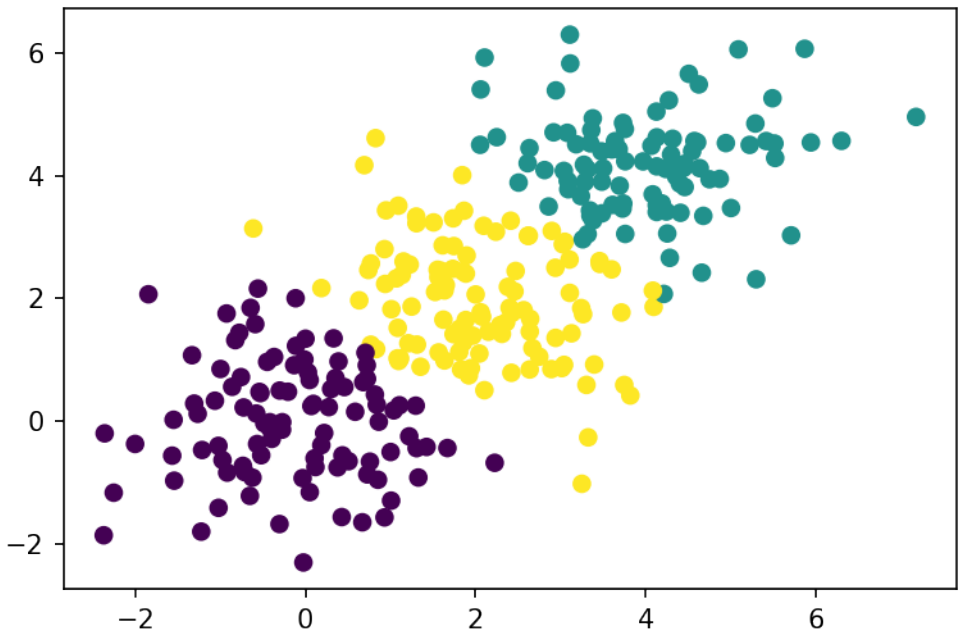
2.1 sklearn 实现

k 均值聚类的接口位于 sklearn.cluster.KMeans, 其文档可见**此处**。其中较为重要的参数有：

- n_clusters：类的个数 k

```
from sklearn.cluster import KMeans

clf = KMeans(n_clusters=3)
clf.fit(X)
```

```
import matplotlib.pyplot as plt
plt.scatter(X[:,0],X[:,1],c=clf.labels_)
plt.show()
```



2.2 k 均值聚类

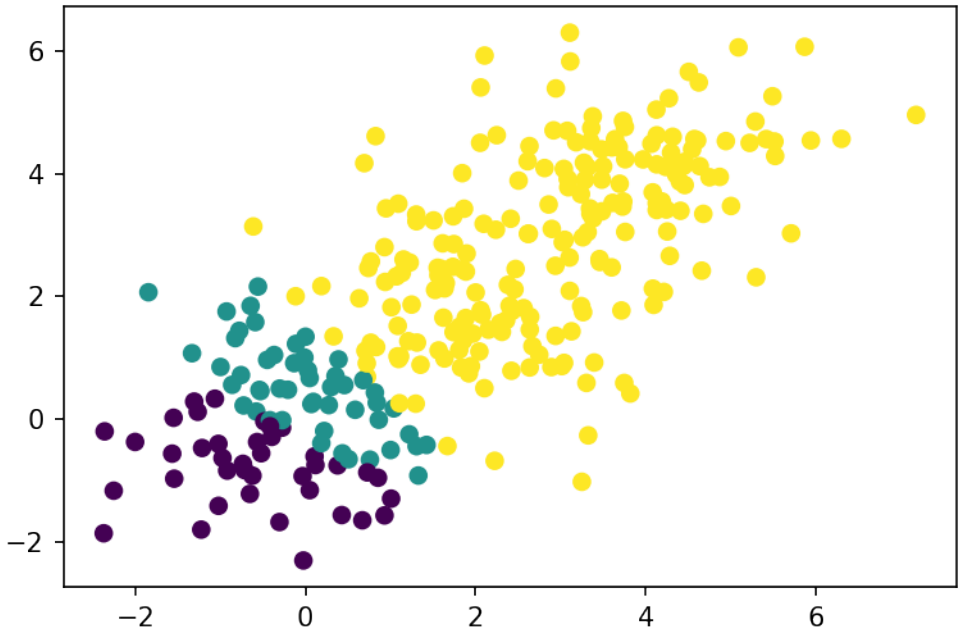
```
class kmeans:

    def __init__(self,k=1):
        self.k = k
```

```
    def fit(self,X):
        n,_ = X.shape
        ## initial
        idx = np.random.choice([i for i in range(len(X))],3)
        center = X[idx,:]
        ## iteration
        dist = np.zeros((n,self.k))
        old_center = center+1
        while np.mean(np.abs(old_center-center))>1e-4:
            ## partition
            for k in range(self.k):
                dist[:,k] = np.sum((X - center[k,:])**2,1)
            label = np.argmin(dist,1)
            ## calculate center
            old_center = center
            for k in range(self.k):
                center[k,:] = np.mean(X[label==k,:],0)
        self.center = center
        self.label = label


    def predict(self,new_X):
        n,_ = new_X.shape
        dist = np.zeros((n,self.k))
        for k in range(self.k):
            dist[:,k] = np.sum((new_X - center[k,:])**2,1)
        return np.argmin(dist,1)
```

使用该方法进行聚类时可能会达到局部极小值。一个改进的方法是使用不同的随机的初始点进行多次拟合，并从中选取效果最好的一个。其中，“效果最好的” 可以根据损失函数的大小进行选择。

此处只进行了一次初值的随机选取，效果如下，可以发现其陷入了局部极小值：

```
clf = kmeans(3)
clf.fit(X)
plt.scatter(X[:,0],X[:,1],c=clf.label)
```


6
4
2
0
−2
−2
0
2
4
6

主成分分析

1 主成分分析

1.1 总体主成分

主成分分析使用正交变换将线性相关的观测数据转换为少数几个线性无关变量，同时使得少数几个线性无关的变量可以尽可能的提取出原始变量的信息。主成分分析使用方差来度量变量的信息量。

假设数据为 $x=(x_1,x_2,...,x_m)^T$，其均值为 $\mu=E(x)=(\mu_1,\mu_2,...,\mu_m)$，协方差矩阵为 $\Sigma=cov(x,x)=E[(x-\mu)(x-\mu)^T]$。主成分分析使用线性变换改变原始数据：$y_i=\alpha_i^T x, i=1,2,...,m$，其中 α_i 需要满足以下三个条件：

1. 系数向量 α_i 是单位向量，即 $\alpha_i^T\alpha_i=1, i=1,2,...,m$
2. 变量 y_i 和 y_j 互不相关，即 $cov(y_i,y_j)=0(i\neq j)$
3. 变量 y_1 是 x 所有线性变换中方差最大的；y_k 是与 $y_{k-1},y_{k-2},...,y_1$ 都不相关的 x 的所有线性变换中方差最大的。

我们将 $y_i,y_2,...,y_m$ 分别称为第一主成分，第二主成分直至第 m 主成分。主成分可以通过以下方式求得：

首先求第一主成分 $y_1=\alpha_1^T x$。由于 $Var(y_1)=\alpha_1^T\Sigma\alpha_1$，所以易知该问题和以下带约束优化问题相同：

$$\begin{aligned}\max_{\alpha_1} \quad & \alpha_1^T\Sigma\alpha_1 \\ s.t. \quad & \alpha_1^T\alpha_1=1\end{aligned}$$

构建拉格朗日函数 $L(\alpha_1,\lambda)=\alpha_1^T\Sigma\alpha_1-\lambda(\alpha_1^T\alpha_1-1)$。对 α_1 求导，并令其等于 0，可以得到：

$$\frac{\partial L}{\partial\alpha_1}=2\Sigma\alpha_1-2\lambda\alpha_1=0$$

因此，λ 是 Σ 的特征值，α_1 是其对应的特征向量。因此目标函数可以简化为 $\alpha_1^T\Sigma\alpha_1=\alpha_1^T\lambda\alpha_1=\lambda\alpha_1^T\alpha_1=\lambda$。因此，此处 λ 应为 Σ 的最大特征值 λ_1，α_1 为其对应的特征向量。

x 的第二主成分同样可以转化为一个带约束的优化问题。注意到 $0=Cov(y_1,y_2)=Cov(\alpha_1x,\alpha_2x)=\alpha_2^T\Sigma\alpha_1=\alpha_2^T\lambda\alpha_1$，因此该优化问题有约束条件 $\alpha_2^T\alpha_1=\alpha_1^T\alpha_2=0$：

$$\begin{aligned}\max_{\alpha_2} \quad & \alpha_2^T \Sigma \alpha_2 \\ s.t. \quad & \alpha_2^T \alpha_2 = 1 \\ & \alpha_2^T \alpha_1 = \alpha_1^T \alpha_2 = 0\end{aligned}$$

构建拉格朗日函数 $L(\alpha_2, \lambda, \phi) = \alpha_2^T \Sigma \alpha_2 - \lambda(\alpha_2^T \alpha_2 - 1) - \phi \alpha_2^T \alpha_1$。对 α_2 求导，并令其等于 0，有：

$$\frac{\partial L}{\partial \alpha_2} = 2\Sigma\alpha_2 - 2\lambda\alpha_2 - \phi\alpha_1 = 0$$

左乘 α_1^T，有 $0 = 2\alpha_1^T \Sigma \alpha_2 - 2\lambda \alpha_1^T \alpha_2 - \phi \alpha_1^T \alpha_1 = -\phi$。因此 $\phi = 0$，代入上式，有 $\Sigma\alpha_2 - \lambda\alpha_2 = 0$。代入目标函数，同样可以得到 α_2 是 Σ 的第二大特征值 λ_2 对应的单位特征向量。

以此类推，可以得到：x 的第 k 主成分是 $\alpha_k^T x$，且 $Var(\alpha_k^T x) = \lambda_k$。此处 λ_k 是 Σ 的第 k 大的特征值，α_k 是其对应的特征向量。

总体主成分具有以下性质：

1. 总体主成分 y 的协方差矩阵是对角矩阵：$Cov(y) = \Lambda = diag(\lambda_1, \lambda_2, ..., \lambda_m)$
2. 总体主成分 y 的方差之和等于随机变量 x 的方差之和：$\sum_{i=1}^m \lambda_i = \sum_{i=1}^m \sigma_{ii}$
3. 第 k 个主成分 y_k 与变量 x_i 的相关系数（称为因子负荷量）为：$\rho(y_k, x_i) = \frac{\sqrt{\lambda_k}\alpha_{ik}}{\sqrt{\sigma_{ii}}}$

主成分分析是一种降维方法，其通过一个线性变换将高维数据压缩为低维数据。可以证明，如果使用线性变换将高维数据压缩为低维数据，主成分分析可以压缩后的数据的方差之和最大。

在使用主成分分析对数据进行降维时，还有以下几个值得注意的地方：

1. 对于主成分的个数的选择：往往使用累计方差贡献率来确定主成分个数。前 k 个主成分的累计贡献率为 $\eta_k = (\sum_{i=1}^k \lambda_i)/(\sum_{i=1}^m \lambda_i)$。一般选择 k 使得 η_k 达到 70% 至 80% 左右
2. 为了防止变量的量纲对求解主成分产生不利影响，往往需要对各个变量进行标准化：$x_i^* = \frac{x_i - E(x_i)}{\sqrt{var(x_i)}}$，并使用 x_i^* 继续进行主成分分析

1.2 样本主成分

在实际应用中，需要在实际观测数据上进行主成分分析，称为样本主成分。样本主成分作用于样本协方差阵 S 或样本相关矩阵 R。

$$S = \frac{1}{n-1}\sum_{i=1}^n (x_i - \bar{x})(x_i - \bar{x})^T$$

$$R = [r_{ij}]_{m\times m}, r_{ij} = \frac{s_{ij}}{\sqrt{s_{ii}s_{jj}}}$$

在得到 S 或 R 后，便可对矩阵进行分解以求得样本主成分，具体步骤如下：

输入：数据矩阵 X

1. 根据数据特征计算样本协方差阵 S 或样本相关矩阵 R
2. 对 S 或 R 进行矩阵分解得到其特征值 $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_m$ 和对应的单位特征向量 $\alpha_1, \alpha_2, ..., \alpha_m$
3. 根据方差贡献率选择主成分个数 k
4. 求出 k 个主成分 $y_i = \alpha_i^T x, i = 1, 2, ..., k$

输出：求出的 k 个主成分 $y_1, y_2, ..., y_k$。

该步骤是从数据的协方差阵或相关矩阵出发，对其进行方阵分解求得主成分。事实上，也可直接从数据矩阵出发，直接进行奇异值分解得到主成分。该做法是：对于矩阵 A，求其截断奇异值分解 $A \approx U_k \Sigma_k V_k^T$，并通过奇异值分解得到主成分。具体做法如下：

输入：$m \times n$ 样本矩阵 X，需要对其进行中心化

1. 若需要进行标准化，令 $X^* = \frac{1}{\sqrt{n-1}} X^T$；若不需要标准化，令 $X^* = X^T$
2. 根据主成分个数 k 对 X^* 进行奇异值分解 $X^* = U_k \Sigma_k V_k^T$
3. 计算前 k 个主成分矩阵 $Y = V^T X$

输出：主成分矩阵 Y。

2 代码实现

本次考虑的数据集为 iris 数据集，其可在 sklearn.datasets 中得到。

2.1 sklearn 实现

准备数据：

```
import numpy as np
from sklearn import datasets
X = datasets.load_iris()['data']
```

主成分分析的接口位于 sklearn.decomposition.PCA, 其文档可见**此处**。其中较为重要的参数有：

- n_components：主成分个数
- whiten：是否对数据标准化

此处数据集的维度是 4，使用主成分分析时，设置主成分的个数为 2.

```
from sklearn.decomposition import PCA
```

```
clf = PCA(n_components=2)
clf.fit(X)
```

2 个主成分的累计贡献率如下所示：

```
print(np.cumsum(clf.explained_variance_ratio_))
```

```
## [0.92461872 0.97768521]
```

可以使用以下语句得到变换后的主成分：

```
Y = clf.fit_transform(X)
```

2.2 主成分分析

```
class Pca:

    def __init__(self,n_components=1,standard=True):
        self.n_components = n_components
        self.standard = standard

    def fit(self,X):
        if self.standard:
            covx = np.corrcoef(X.T)
        else:
            covx = np.cov(X.T)
        u,v = np.linalg.eig(covx)
        self.variance = u
        self.variance_ratio = u/np.sum(u)
        self.trans = v

    def fit_transform(self,X):
        return X.dot(self.trans[:,:self.n_components])
```

结果如下：

```
clf = Pca(n_components=2,standard=False)
clf.fit(X)
print(np.cumsum(clf.variance_ratio))
```

```
## [0.92461872 0.97768521 0.99478782 1.        ]
```

同样，可如下得到变换后的主成分：

```
Y = clf.fit_transform(X)
```

潜在语义分析

1 潜在语义分析

1.1 单词向量空间

给定一个含有 n 个文本的集合 $D = \{d_1, d_2, ..., d_n\}$ 以及在所有文本中出现的 m 个单词的集合 $W = \{w_1, w_2, ..., w_m\}$。则单词在文本中出现的数据可以用一个**单词-文本矩阵**表示，记作 $X = [x_{ij}]_{m \times n}$。元素 x_{ij} 表示单词 w_i 在文本 d_j 中出现的频数或权值。单词-文本矩阵是一个稀疏矩阵。

权值通常用单词频率-逆文本频率 (TF-IDF) 表示，其定义为：

$$TFIDF_{ij} = TF_{ij}IDF_{ij} = \frac{tf_{ij}}{tf_{\cdot j}} \log \frac{df}{df_i}$$

其中 tf_{ij} 是单词 w_i 在文本 d_j 中的频数，$tf_{\cdot j}$ 是文本 d_j 中所有单词的频数之和，df_i 是含有单词 w_i 的文本数，df 是文本 D 的全部文本数。因此，TF-IDF 是两种重要度的积，表示综合重要度。

在判断文本相似度时，对于文本 d_i 和 d_j 可以直接使用单词-文本矩阵的第 i 列向量 x_i 和第 j 列向量 x_j 的内积 $x_i \cdot x_j$ 或余弦夹角 $\frac{x_i \cdot x_j}{||x_i|| ||x_j||}$ 进行判断。该模型简单实用，但不能很好地处理一词多义性和多词一义性。

1.2 话题向量空间

为处理一词多义性和多词一义性，可以引入话题向量空间。话题可以由若干个语义相关的单词表示。假设所有文本共有 k 个话题，假设每个话题由一个定义在单词集合 W 上的 m 维向量表示，称为话题向量，即 $t_l = (t_{1l}, t_{2l}, ..., t_{ml})^T, l = 1, 2, ..., k$。其中，$t_{il}$ 是单词 w_i 在话题 t_l 上的权值。k 个话题向量构成一个话题向量空间 T。话题向量空间也可以由**单词-话题矩阵**表示，记作 $T = [t_{ij}]_{m \times k}$。

对于文本集合 D 中的文本 d_j，其在单词向量空间中的表示为 x_j，将其投影到话题向量空间 T 中，可以得到在话题向量空间中的表示 $y_j = (y_{1j}, y_{2j}, ..., y_{kj})^T$。其中 y_{ij} 是文本 d_j 在话题 t_l 上的权值。记 $Y = [y_1, y_2, ..., y_n]$ 为**话题-文本矩阵**。

考虑使用线性变换将在单词向量空间中的文本向量 x_j 通过它在话题空间中的向量 y_j 近似表示，即将 k 个话题向量以 y_j 为系数进行线性组合近似表示。

$$x_j \approx y_{1j}t_1 + y_{2j}t_2 + ... + y_{kj}t_k, \quad j = 1, 2, ..., n$$

所以，**单词-文本矩阵 X 可以近似地表示为单词-话题矩阵 T 和话题-文本矩阵 Y 的乘积**，即 $X \approx TY$。其中仅有单词-文本矩阵 X 是可以观测的。潜在语义分析就是通过可观测的 X，根据数据内在的关联信息，同时决定话题空间 T 和文本在话题空间的表示 Y。

1.3 矩阵奇异值分解法

矩阵奇异值分解法使用截断奇异值分解对单词-文本矩阵 X 进行分解。

$$X \approx U_k \Sigma_k V_k^T$$

式中 X 是 $m \times n$ 的单词-文本矩阵。$k \leq n \leq m$，U_k 是 $m \times k$ 矩阵，它的列由 X 的前 k 个互相正交的左奇异向量组成。Σ_k 是 k 阶对角方阵，对角元素是 X 的前 k 个最大的奇异值。V_k 是 $n \times k$ 矩阵，它的列由 X 的前 k 个互相正交的右奇异向量组成。

矩阵奇异值分解法令

$$X \approx U_k \Sigma_k V_k^T = U_k(\Sigma_k V_k^T) := TY$$

即将 U_k 作为话题空间，其每一列表示一个话题；$\Sigma_k V_k^T$ 作为文本在话题向量空间中的表示，其第 j 列 $(\Sigma_k V_k^T)_j$ 表示文本 d_j 在话题向量空间中的表示。矩阵奇异值方法分解法步骤如下：

输入：单词-文本矩阵 X 和话题个数 $k \leq n \leq m$。

1. 对 X 进行截断奇异值分解 $X \approx U_k \Sigma_k V_k^T$
2. 计算话题空间 $T = U_k$ 和文本在话题向量空间的表示 $Y = \Sigma_k V_k^T$

输出：话题空间 T 和文本在话题向量空间的表示 Y。

使用矩阵奇异值分解法计算本文相似度时，可直接用 $(\Sigma_k V_k^T)_j$ 进行内积或余弦夹角的计算。

1.4 非负矩阵分解法

对于一个非负矩阵（矩阵中的所有元素非负）$X \geq 0$，可以找到两个非负矩阵 $W \geq 0$ 和 $H \geq 0$，使得 $X \approx WH$。X 是一个 $m \times n$，假设 $k < \min(m, n)$，非负矩阵 W, H 分别为 $m \times k, k \times n$ 的矩阵。

使用非负矩阵分解法可以找到单词-文本矩阵的另一种分解方式。即 W 为话题向量空间，H 是文本在话题向量空间中的表示。

非负矩阵分解法等价于寻找 W, H，而这又可以转化为一个优化问题，可以分别使用平方损失和散度损失来刻画这一问题。平方损失的优化问题如下：

$$\min_{W,H} \quad ||X-WH||^2 = \sum_{i,j}(X_{ij}-(WH)_{ij})^2$$
$$s.t. \quad W,H \geq 0$$

散度损失的优化问题如下：

$$\min_{W,H} \quad D(X||WH) = \sum_{i,j}(X_{ij}\log\frac{X_{ij}}{(WH)_{ij}} - X_{ij} + (WH)_{ij})$$
$$s.t. \quad W,H \geq 0$$

两个问题可分别通过以下参数更新方法求得参数。对于平方损失 $||X-WH||^2$，更新方法为

$$H_{lj} := H_{lj}\frac{(W^TX)_{lj}}{(W^TWH)_{lj}}$$

$$W_{il} := W_{il}\frac{(XH^T)_{il}}{(WHH^T)_{il}}$$

对于散度损失，更新方法为：

$$H_{lj} := H_{lj}\frac{\sum_i[W_{il}X_{ij}/(WH)_{ij}]}{\sum_i W_{il}}$$
$$W_{il} := W_{il}\frac{\sum_j H_{lj}X_{ij}/(WH)_{ij}}{\sum_j H_{lj}}$$

以上两个更新方法都是梯度下降法的特殊形式，且可以证明，当且仅当 W,H 是损失函数的稳定点时，函数的更新不变。非负矩阵分解法步骤如下：

输入：单词-文本矩阵 $X \geq 0$ 和话题个数 k。

1. 初始化：$W,H \geq 0$，并对 W 的每一列数据归一化
2. 迭代直至收敛：

- 更新 W 的元素，从 $l = 1,2,...,k; i = 1,2,...,m$ 依次更新 W_{il}
- 更新 H 的元素，从 $l = 1,2,...,k; i = 1,2,...,n$ 依次更新 H_{lj}

输出：话题空间 W 和文本在话题向量空间的表示 H。

2 代码实现

此处的数据为：

$$X = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \\ 1 & 2 & 2 & 1 \end{bmatrix}$$

2.1 sklearn 实现

准备数据：

```
import numpy as np
X = np.array([[2,0,0,0],[0,2,0,0],[0,0,1,0],[0,0,2,3],[0,0,0,1],[1,2,2,1]])
```

潜在语义分析的 sklearn 接口位于 sklearn.decomposition.TruncatedSVD, 其文档可见**此处**。注意该接口实际就是截断奇异值分解的接口，官方文档中也有显示 (Dimensionality reduction using truncated SVD (aka LSA).)。其中较为重要的参数有：

- n_components：话题个数

```
from sklearn.decomposition import TruncatedSVD
clf = TruncatedSVD(n_components=2)
clf.fit(X)
```

其话题向量空间为：

```
print(clf.fit_transform(X))
```

```
## [[ 0.3511592   0.78272254]
##  [ 0.7023184   1.56544509]
##  [ 0.63851545 -0.03795798]
##  [ 3.26283385 -1.52321299]
##  [ 0.66193431 -0.48243234]
##  [ 2.81686322  1.39845805]]
```

文本在话题空间的表示为：

```
print(clf.components_)
```

```
## [[ 0.1755796   0.3511592   0.63851545  0.66193431]
##  [ 0.39136127  0.78272254 -0.03795798 -0.48243234]]
```

2.2 矩阵奇异值分解算法

```
class LSA:

    def __init__(self,topic_num=2):
        self.topic_num = topic_num

    def fit(self,X):
        U,Sigma,V = np.linalg.svd(X)
        T,Y = U[:,:self.topic_num],(np.diag(Sigma).dot(V))[:self.topic_num,:]
        self.components = Y
        self.topic_vector = T

    def fit_transfrom(self,X):
        return self.topic_vector
```

其结果为：

```
clf = LSA(topic_num=2)
clf.fit(X)
```

```
print(clf.fit_transfrom(X))
```

```
## [[-0.07843687 -0.28442303]
##  [-0.15687373 -0.56884607]
##  [-0.14262235  0.01379304]
##  [-0.72880467  0.55349991]
##  [-0.14785332  0.17530461]
##  [-0.6291902  -0.50816689]]
```

```
print(clf.components)
```

```
## [[-0.78606393 -1.57212786 -2.85861209 -2.96345752]
##  [-1.07701296 -2.15402591  0.10445908  1.32763745]]
```

注意到这和 sklearn 输出的结果不完全一样，原因是：numpy 中的 svd 分解未对奇异向量组做正交化且矩阵的奇异值分解不唯一。

概率潜在语义分析

1 概率潜在语义分析

1.1 生成模型和共现模型

假定有单词集合 $W = \{w_1, w_2, ..., w_M\}$，文本集合 $D = \{d_1, d_2, ..., d_N\}$ 和话题集合 $Z = \{z_1, z_2, ..., z_K\}$。随机变量 w 取值于单词集合，随机变量 d 取值于文本集合，随机变量 z 随机取值于话题分布。

生成模型按照以下步骤生成文本-单词共现数据：

1. 依照文本的概率分布 $P(d)$，从文本集合中随机选取一个文本 d，共生成 N 个文本；针对每个文本，执行以下操作
2. 在文本 d 给定的条件下，依据条件概率分布 $P(z|d)$，从话题集合中随机选取一个话题 z，共生成 L 个话题
3. 在话题 z 给定的条件下，依据条件概率分布 $P(w|z)$，对每个话题从单词集合中随机选取一个单词 w

在此模型中，单词变量 w 和文本变量 d 是观测向量，话题变量 z 是隐变量。对于数据 $T = (w, d)$，可以得到其生成概率的乘积

$$P(T) = \prod_{(w,d)} P(w,d)^{n(w,d)}$$

其中 $n(w,d)$ 表示 (w,d) 的出现次数。对于其出现概率 $P(w,d)$ 可以表示为：

$$P(w,d) = P(d)P(w|d) = P(d)\sum_z P(w,z|d) = P(d)\sum_z P(z|d)P(w|z)$$

其中最后一个等号是基于假设：在给定话题 z 的条件下，单词 w 与文本 d 条件独立。生成模型的数据生成步骤是 $d \to z \to w$。

共现模型是一种和生成模型等价的模型，与生成模型 $d \to z \to w$ 序列不同，共现模型从话题 z 出发，其同时单词 w 和文本 d，其中生成单词的步骤需要重复多次。因此共现模型的序列为 $w \leftarrow z \to d$，(w,d) 的出现概率为

$$P(w,d)=\sum_z P(z)P(w|z)P(d|z)$$

事实上，上式和潜在语义分析中的 svd 分解十分类似。潜在语义分析中将单词-文本矩阵 X 分解为 $X=U\Sigma V^T$，此处可如下表示：$X=[P(w,d)]_{M\times N}, U=[P(w|z)]_{M\times K}, \Sigma=[P(z)]_{K\times K}, V=[P(d|z)]_{N\times K}$。

1.2 EM 算法

设单词集合为 $W=\{w_1,w_2,...,w_M\}$，文本集合为 $D=\{d_1,d_2,...,d_N\}$，话题集合为 $Z=\{z_1,z_2,...,z_K\}$。给定单词-文本共现数据 $T=\{n(w_i,d_j)\}, i=1,2,...,M; j=1,2,...,N$。为估计概率潜在语义分析模型的参数，构建对数似然函数：

$$\begin{aligned}L&=\sum_{i=1}^M\sum_{j=1}^N n(w_i,d_j)\log P(w_i,d_j)\\&=\sum_{i=1}^M\sum_{j=1}^N n(w_i,d_j)\log\left[\sum_{k=1}^K P(w_i|z_k)P(z_k|d_j)\right]\end{aligned}$$

该对数似然函数中含有隐变量，因此无法直接估计，需要使用 EM 算法进行迭代。可以得到，概率潜在语义分析的 Q 函数是：

$$Q=\sum_{k=1}^K\left\{\sum_{j=1}^N n(d_j)\left[\log P(d_j)+\sum_{i=1}^M\frac{n(w_i,d_j)}{n(d_j)}\log P(w_i|z_k)P(z_k|d_j)\right]\right\}P(z_k|w_i,d_j)$$

其中，$n(d_j)=\sum_{i=1}^M n(w_i,d_j)$ 表示文本 d_j 中的单词个数，$n(w_i,d_j)$ 表示单词 w_i 在文本 d_j 中的出现次数。同时由于式中的 $P(d_j)$ 可以直接从从数据中按比例估计得出，在极大化 Q 时可以不进行考虑。因此，极大化 Q 和极大化 Q^* 相同，Q^* 形式如下：

$$Q^*=\sum_{i=1}^M\sum_{j=1}^N n(w_i,d_j)\sum_{k=1}^K P(z_k|w_i,d_j)\log[P(w_i|z_k)P(z_k|d_j)]$$

式中的 $P(z_k|w_i,d_j)$ 可以通过贝叶斯公式进行计算，有：

$$P(z_k|w_i,d_j)=\frac{P(w_i|z_k)P(z_k|d_j)}{\sum_{k=1}^K P(w_i|z_k)P(z_k|d_j)}$$

在 $P(z_k|w_i,d_j)$ 可通过上一步的 $P(w_i|z_k),P(z_k|d_j)$ 计算的情况下，可以极大化 Q^* 得到新的 $P(w_i|z_k),P(z_k|d_j)$ 的估计。该求解过程需在约束条件 $\sum_{i=1}^M P(w_i|z_k)=1, k=1,2,...,K$ 和 $\sum_{k=1}^K P(z_k|d_j)=1, j=1,2,...,N$ 下进行。通过拉格朗日乘子法，可以得到：

$$P(w_i|z_k) = \frac{\sum_{j=1}^{N} n(w_i, d_j)P(z_k|w_i, d_j)}{\sum_{m=1}^{M}\sum_{j=1}^{N} n(w_m, d_j)P(z_k|w_m, d_j)}$$

$$P(z_k|d_j) = \frac{\sum_{i=1}^{M} n(w_i, d_j)P(z_k|w_i, d_j)}{n(d_j)}$$

因此，概率潜在语义分析模型的步骤可如下所示：

输入：单词集合为 $W = \{w_1, w_2, ..., w_M\}$，文本集合为 $D = \{d_1, d_2, ..., d_N\}$，话题集合为 $Z = \{z_1, z_2, ..., z_K\}$，共现数据 $\{n(w_i, d_j)\}, i = 1, 2, ..., M, j = 1, 2, ..., N$。

1. 初始化参数 $P(w_i|z_k), P(z_k|d_j)$。两者分别有 MK 个和 NK 个
2. 计算
$$P(z_k|w_i, d_j) = \frac{P(w_i|z_k)P(z_k|d_j)}{\sum_{k=1}^{K} P(w_i|z_k)P(z_k|d_j)}$$
3. 更新参数：
$$P(w_i|z_k) = \frac{\sum_{j=1}^{N} n(w_i, d_j)P(z_k|w_i, d_j)}{\sum_{m=1}^{M}\sum_{j=1}^{N} n(w_m, d_j)P(z_k|w_m, d_j)}$$
$$P(z_k|d_j) = \frac{\sum_{i=1}^{M} n(w_i, d_j)P(z_k|w_i, d_j)}{n(d_j)}$$
4. 重复 2,3，直至收敛

输出：参数 $P(w_i|z_k)$ 和 $P(z_k|d_j)$。

2 代码实现

概率潜在语义分析在 sklearn 中没有直接实现的函数，此处直接对其进行实现。本次考虑的数据集如下：

Index Words	Titles								
	T1	T2	T3	T4	T5	T6	T7	T8	T9
book			1	1					
dads						1			1
dummies		1						1	
estate							1		1
guide	1					1			
investing	1	1	1	1	1	1	1	1	1
market	1		1						
real							1		1
rich						2			1
stock	1		1					1	
value				1	1				

准备数据：

```
import numpy as np

X = np.array([[0,0,1,1,0,0,0,0,0],
              [0,0,0,0,0,1,0,0,1],
              [0,1,0,0,0,0,0,1,0],
              [0,0,0,0,0,0,1,0,1],
              [1,0,0,0,0,1,0,0,0],
              [1,1,1,1,1,1,1,1,1],
              [1,0,1,0,0,0,0,0,0],
              [0,0,0,0,0,0,1,0,1],
              [0,0,0,0,0,2,0,0,1],
              [1,0,1,0,0,0,0,1,0],
              [0,0,0,1,1,0,0,0,0]])
```

概率潜在语义分析模型可按如下实现：

```
class PLSA:

    def __init__(self,n_components=1):
        self.n_components = n_components

    def fit(self,X):
        M,N = X.shape
```

```
        K = self.n_components

        ## initialization
        wz = np.random.random((M,K))
        zd = np.random.random((K,N))
        wz = wz/np.sum(wz,0).reshape(1,-1)
        zd = zd/np.sum(zd,0).reshape(1,-1)

        error = 1
        while error>1e-3:
            ## calculate P(z|w,d)
            zwd = np.array([zd[:,i].reshape(-1)*wz for i in range(zd.shape[1])])
            zwd = zwd/np.sum(zwd,2).reshape(N,M,1)
            ## update P(w|z) and P(z|d)
            zwd = zwd*X.T.reshape(N,M,1)
            wz_new = np.sum(zwd,axis=0)
            wz_new = wz_new/np.sum(wz_new,0).reshape(1,-1)
            zd_new = np.sum(zwd,axis=1).T
            zd_new = zd_new/np.sum(zd_new,0).reshape(1,-1)

            error = np.mean(np.abs(zd_new-zd))+np.mean(np.abs(wz_new-wz))
            wz,zd = wz_new,zd_new

        self.wz = wz
        self.zd = zd
```

其结果如下：

```
clf = PLSA(n_components=3)
clf.fit(X)
print('单词-话题分布：\n',np.around(clf.wz,3))
```

```
## 单词-话题分布：
##  [[0.    0.    0.154]
##  [0.    0.201 0.   ]
##  [0.249 0.    0.   ]
##  [0.126 0.099 0.   ]
##  [0.    0.1   0.077]
##  [0.376 0.198 0.308]
##  [0.    0.    0.154]
```

```
##  [0.126 0.099 0.   ]
##  [0.    0.302 0.   ]
##  [0.124 0.    0.154]
##  [0.    0.    0.154]]
```

```
print('话题-文本分布：\n',np.around(clf.zd,3))
```

```
## 话题-文本分布：
##  [[0.    1.    0.    0.    0.    0.    0.995 1.    0.012]
##  [0.    0.    0.    0.    0.    0.997 0.005 0.    0.988]
##  [1.    0.    1.    1.    1.    0.003 0.    0.    0.   ]]
```

潜在狄利克雷分配

1 潜在狄利克雷分配

1.1 狄利克雷分布

若多元连续随机变量 $\theta = (\theta_1, \theta_2, ..., \theta_k)$ 的概率密度函数为

$$f(\theta|\alpha) = \frac{\Gamma\left(\sum_{i=1}^{k} \alpha_i\right)}{\prod_{i=1}^{k} \Gamma(\alpha_i)} \prod_{i=1}^{k} \theta_i^{\alpha_i - 1}$$

其中 $\sum_{i=1}^{k} \theta_i = 1, \theta_i \geq 0, \alpha = (\alpha_1, \alpha_2, ..., \alpha_k), \alpha_i > 0, i = 1, 2, ..., k$，则称随机变量 θ 服从参数为 α 的狄利克雷分布，记作 $\theta \sim Dir(\alpha)$。

令

$$B(\alpha) = \frac{\prod_{i=1}^{k} \Gamma(\alpha_i)}{\Gamma\left(\sum_{i=1}^{k} \alpha_i\right)}$$

$B(\alpha)$ 是规范化因子，称为多元贝塔函数。可将狄利克雷分布的概率密度函数记为

$$f(\theta|\alpha) = \frac{1}{B(\alpha)} \prod_{i=1}^{k} \theta_i^{\alpha_i - 1}$$

假设随机变量 X 服从多项分布 $X \sim Multi(n, \theta), n = (n_1, n_2, ..., n_k), \theta = (\theta_1, \theta_2, ..., \theta_k)$，则其概率密度函数为

$$f(X|\theta) = \theta_1^{n_1} \theta_2^{n_2} ... \theta_k^{n_k} = \prod_{i=1}^{k} \theta_i^{n_i}$$

其中 X 的参数 θ 满足的先验分布为狄利克雷分布 $f(\theta|\alpha)$，参数为 $\alpha = (\alpha_1, \alpha_2, ..., \alpha_k)$。因此参数 θ 的后验分布为：

$$
\begin{aligned}
f(\theta|X,\alpha) &= \frac{f(X|\theta)f(\theta|\alpha)}{f(X|\alpha)} \\
&= \frac{\prod_{i=1}^{k}\theta_i^{n_i}\frac{1}{B(\alpha)}\theta_i^{\alpha_i-1}}{\int\prod_{i=1}^{k}\theta_i^{n_i}\frac{1}{B(\alpha)}\theta_i^{\alpha_i-1}d\theta} \\
&= \frac{1}{B(\alpha+n)}\prod_{i=1}^{k}\theta_i^{\alpha_i+n_i-1} \\
&= Dir(\theta|\alpha+n)
\end{aligned}
$$

因此，如果多项分布的先验分布是狄利克雷分布，则其后验分布也是狄利克雷分布。称狄利克雷分布是多项分布的**共轭先验**。

1.2 模型定义

潜在狄利克雷分配使用三个集合：一是**单词集合**$W = \{w_1, w_2, ..., w_V\}, V$ 是单词的个数。二是**文本集合**$D = \{d_1, d_2, ..., d_M\}, M$ 是文本数量；文本 $d_m = (w_{m1}, w_{m2}, ..., w_{mN_m})$ 是一个单词序列，N_m 是文本 d_m 中的单词个数。三是**话题集合**$Z = \{z_1, z_2, ..., z_K\}, K$ 是话题个数。

由话题 z_k 生成单词是由其条件分布 $f(w|z_k)$ 决定，服从多项分布，参数为 $\phi_k = (\phi_{k1}, \phi_{k2}, ..., \phi_{kV})$，该参数服从一个超参数为 β 的狄利克雷分布。所有话题的参数向量构成一个 $K \times V$ 矩阵 $\phi = \{\phi_k\}_{k=1}^{K}$，超参数 β 也是一个 V 维向量 $\beta = (\beta_1, \beta_2, ..., \beta_V)$。

由话题生成文本 d_m 是由其条件分布 $f(z|d_m)$ 决定，服从多项分布，参数为 $\theta_m = (\theta_{m1}, \theta_{m2}, ..., \theta_{mK})$，该参数服从一个超参数为 α 的狄利克雷分布。所有话题的参数向量构成一个 $M \times K$ 矩阵 $\theta = \{\theta_m\}_{m=1}^{M}$，超参数 α 也是一个 K 维向量 $\alpha = (\alpha_1, \alpha_2, ..., \alpha_K)$。

因此，每一个文本 d_m 中的每一个单词 w_{mn} 由该文本的话题分布 $f(z|d_m)$ 和所有话题的单词分布 $f(w|z_k)$ 决定。潜在狄利克雷分配的生成算法如下：

输入：单词集合 W，文本集合 D，话题集合 W 和狄利克雷分布的超参数 α, β。

1. 生成话题的单词分布：对于话题 $z_k, k = 1, 2, ..., K$，生成多项分布的参数 $\phi_k \sim Dir(\beta)$，作为话题的单词分布 $f(w|z_k)$
2. 生成文本的话题分布：对于文本 $d_m, m = 1, 2, ..., M$，生成多项分布的参数 $\theta_m \sim Dir(\alpha)$，作为文本的话题分布 $f(z|d_m)$
3. 按照多项分布 $Multi(\theta_m)$ 随机生成一个话题 $z_{mn} \sim Multi(\theta_m), m = 1, 2, ..., M; n = 1, 2, ..., N_m$
4. 按照多项分布 $Multi(\phi_{z_{mn}})$ 随机生成一个单词 $w_{mn} \sim Multi(\phi_{z_{mn}}), m = 1, 2, ..., M; n = 1, 2, ..., N_m$

输出：生成的文本 $\{d_i = \{w_{m1}, w_{m2}, ..., w_{mN_m}\}\}_{m=1}^{M}$。

以概率图模型的视角，LDA 的图模型为 $\alpha \to \theta_m \to z_{mn} \to w_{mn} \leftarrow \phi_k \leftarrow \beta$。LDA 模型整体是由观测变量和隐变量组成的联合概率分布为

$$f(d,z,\theta,\phi|\alpha,\beta)=\prod_{k=1}^{K}f(\phi_k|\beta)\prod_{m=1}^{M}f(\theta_m|\alpha)\prod_{n=1}^{N_m}f(z_{mn}|\theta_m)f(w_{mn}|z_{mn},\phi)$$

其中，第 m 个文本的联合概率分布为

$$f(d_m,z_m,\theta_m,\phi|\alpha,\beta)=\prod_{k=1}^{K}f(\phi_k|\beta)f(\theta_m|\alpha)\prod_{n=1}^{N_m}f(z_{mn}|\theta_m)f(w_{mn}|z_{mn},\phi)$$

为得到关于 d 的概率分布，先求 d_m 关于 θ_m,ϕ 的分布：

$$f(d_m|\theta_m,\phi)=\prod_{n=1}^{N_m}\left[\sum_{k=1}^{K}P(z_{mn}=k|\theta_m)f(w_{mn}|\phi_k)\right]$$

所以，超参数 α,β 给定下第 m 个文本的生成概率为：

$$f(d_m|\alpha,\beta)=\prod_{k=1}^{K}\int f(\phi_k|\beta)\left[\int f(\theta_m|\alpha)\prod_{n=1}^{N_m}\left[\sum_{k=1}^{K}P(z_{mn}=k|\theta_m)f(w_{mn}|\phi_k)\right]d\theta_m\right]d\phi_k$$

所以，超参数 α,β 给定下所有文本的生成概率为：

$$f(d|\alpha,\beta)=\prod_{k=1}^{K}\int f(\phi_k|\beta)\left[\prod_{m=1}^{M}\int f(\theta_m|\alpha)\prod_{n=1}^{N_m}\left[\sum_{k=1}^{K}P(z_{mn}=k|\theta_m)f(w_{mn}|\phi_k)\right]d\theta_m\right]d\phi_k$$

1.3 LDA 的吉布斯抽样算法

首先介绍**吉布斯抽样**。吉布斯抽样是马尔可夫链蒙特卡洛方法 (MCMC) 中的一种方法，其可用于多元联合分布的抽样和估计。该算法如下：

输入：目标概率分布的密度函数 $p(x)$，函数 $f(x)$，收敛步数 m 和迭代步数 n。

1. 初始化：给出初始样本 $x^{(0)}=\{x_1^{(0)},x_2^{(0)},...,x_k^{(0)}\}^T$
2. 对 i 循环执行。此时，第 $i-1$ 次迭代结束后的样本为 $x^{(i-1)}=\{x_1^{(i-1)},x_2^{(i-1)},...,x_k^{(i-1)}\}^T$，不断执行以下操作：

- 从分布 $p(x_1|x_2^{(i-1)},...,x_k^{(i-1)})$ 抽取 $x_1^{(i)}$
- …
- 从分布 $p(x_j|x_1^{(i)},...,x_{j-1}^{(i)},x_{j+1}^{(i-1)}...,x_k^{(i-1)})$ 抽取 $x_j^{(i)}$

- ...
- 从分布 $p(x_k|x_1^{(i)},...,x_{k-1}^{(i)})$ 抽取 $x_k^{(i)}$

3. 得到样本集合 $\{x^{(m+1)},x^{m+2},...,x^{(n)}\}$
4. 计算结果 $f_{mn}=\frac{1}{n-m}\sum_{i=m+1}^{n}f(x^{(i)})$

输出：估计结果 f_{mn}。

记文本中的单词总集合为 $w=(w_{11},w_{12},...,w_{1N_1},w_{21},w_{22},...,w_{2N_2},...,w_{M1},w_{M2},...,w_{MN_M})$。话题集合是 $z=(z_1,z_2,...,z_M),z_m=(z_{m1},z_{m2},...,z_{mN_m}),m=1,2,...,M$。文本的话题分布和话题的单词分布参数分别为 $\theta=\{\theta_1,\theta_2,...,\theta_M\}$ 和 $\phi=\{\phi_1,\phi_2,...,\phi_K\}$。在超参数 α,β 已知的情况下，需要对联合概率分布 $p(w,z,\theta,\phi|\alpha,\beta)$ 进行估计，其中 w 是观测变量，z,θ,ϕ 是隐变量。

LDA 模型采用*收缩的吉布斯抽样方法*，基本思想是：

1. 首先对隐变量 θ,ϕ 积分，得到边缘概率分布 $p(w,z|\alpha,\beta)$
2. 转换为对不可观测的变量 z 的抽样，按后验分布 $p(z|w,\alpha,\beta)$ 进行吉布斯抽样
3. 得到分布 $p(z|w,\alpha,\beta)$ 的样本集合，使用该集合估计参数 θ,ϕ 的估计值

可以发现，对参数 θ,ϕ 的估计主要需要计算后验概率 $p(z|w,\alpha,\beta)$。由于

$$p(z|w,\alpha,\beta)=\frac{p(w,z|\alpha,\beta)}{p(w|\alpha,\beta)}\propto p(w,z|\alpha,\beta)$$

$p(w|\alpha,\beta)$ 中均是已知变量，可以不予考虑，所以可以考虑 $p(w,z|\alpha,\beta)$，而该概率分布可以进一步分解为

$$p(w,z|\alpha,\beta)=p(w|z,\alpha,\beta)p(z|\alpha,\beta)=p(w|z,\beta)p(z|\alpha)$$

对两个因子 $p(w|z,\beta)$ 和 $p(z|\alpha)$ 分别进行处理。对于 $p(w|z,\beta)$，首先

$$p(w|z,\phi)=\prod_{k=1}^{K}\prod_{v=1}^{V}\phi_{kv}^{n_{kv}}$$

式中，ϕ_{kv} 是第 k 个话题生成单词集合中第 v 个单词的概率，n_{kv} 是第 k 个话题生成单词集合中第 v 个单词的次数。于是

$$
\begin{aligned}
p(w|z,\beta) &= \int p(w|z,\phi)p(\phi|\beta)d\phi \\
&= \int \prod_{k=1}^{K}\prod_{v=1}^{V} \phi_{kv}^{n_{kv}} \frac{1}{B(\beta)} \phi_{kv}^{\beta_v - 1} d\phi \\
&= \prod_{k=1}^{K} \frac{1}{B(\beta)} \int \prod_{v=1}^{V} \phi_{kv}^{n_{kv}+\beta_v-1} d\phi \\
&= \prod_{k=1}^{K} \frac{B(n_k+\beta)}{B(\beta)}
\end{aligned}
$$

其中 $n_k = \{n_{k1}, n_{k2}, ..., n_{kV}\}$。第二个因子 $p(z|\alpha)$ 也可通过类似的方法进行计算，由于

$$
p(z|\theta) = \prod_{m=1}^{M}\prod_{k=1}^{K} \theta_{mk}^{n_{mk}}
$$

式中，θ_{mk} 是第 m 个文本生成第 k 个话题的概率，n_{kv} 是第 m 个文本生成第 k 个话题的次数。于是

$$
\begin{aligned}
p(z|\alpha) &= \int p(z|\theta)p(\theta|\alpha)d\theta \\
&= \int \prod_{m=1}^{M}\prod_{k=1}^{K} \theta_{mk}^{n_{mk}} \frac{1}{B(\alpha)} \theta_{mk}^{\alpha_k - 1} d\theta \\
&= \prod_{m=1}^{M} \frac{1}{B(\alpha)} \int \prod_{k=1}^{K} \theta_{mk}^{n_{mk}+\alpha_k-1} d\theta \\
&= \prod_{m=1}^{M} \frac{B(n_m+\alpha)}{B(\alpha)}
\end{aligned}
$$

其中 $n_m = \{n_{m1}, n_{m2}, ...n_{mK}\}$。联立两式有

$$
p(z,w|\alpha,\beta) = \prod_{k=1}^{K} \frac{B(n_k+\beta)}{B(\beta)} \prod_{m=1}^{M} \frac{B(n_m+\alpha)}{B(\alpha)}
$$

因此，收缩的吉布斯抽样分布的公式为

$$
p(z|w,\alpha,\beta) \propto \prod_{k=1}^{K} \frac{B(n_k+\beta)}{B(\beta)} \prod_{m=1}^{M} \frac{B(n_m+\alpha)}{B(\alpha)}
$$

根据该联合密度函数，可以对 z 进行吉布斯抽样。由于 $p(z|w,\alpha,\beta)$ 是满条件分布，因此该函数的吉布斯抽样分布可以写成

$$
p(z_i|z_{-i},w,\alpha,\beta) = \frac{1}{Z_{z_i}} p(z|w,\alpha,\beta)
$$

其中此处的 i 可以取到所有单词，$z_{-i} = \{z_j : j \neq i\}$，$Z_{z_i}$ 是规范化因子，使左端可以变成一个概率密度函数。由此可以推出：

$$p(z_i|z_{-i}, w, \alpha, \beta) \propto \frac{n_{kv} + \beta_v}{\sum_{v=1}^{V}(n_{kv} + \beta_v)} \frac{n_{mk} + \alpha_k}{\sum_{k=1}^{K}(n_{mk} + \alpha_k)}$$

通过该函数的吉布斯抽样，可以得到一系列关于话题 z 的样本，从而估计参数 $\theta = \{\theta_m\}$ 和 $\phi = \{\phi_k\}$。可以由共轭先验，写出 θ, ϕ 的后验分布。

$$p(\theta_m|z_m, \alpha) = \frac{1}{Z_{\theta_m}} \prod_{n=1}^{N_m} p(z_{mn}|\theta_m) p(\theta_m|\alpha) \sim Dir(\theta_m|n_m + \alpha)$$

$$p(\phi_k|z, w, \beta) = \frac{1}{Z_{\phi_k}} \prod_{i=1}^{I} p(w_i|\phi_k) p(\phi_k|\beta) \sim Dir(\phi_k|n_k + \beta)$$

使用极大似然估计，可以得到：

$$\theta_{mk} = \frac{n_{mk} + \alpha_k}{\sum_{k=1}^{K}(n_{mk} + \alpha_k)}, m = 1, 2, ..., M; k = 1, 2, ..., K$$

$$\phi_{kv} = \frac{n_{kv} + \beta_v}{\sum_{v=1}^{V}(n_{kv} + \beta_v)}, k = 1, 2, ..., K; v = 1, 2, ..., V$$

在算法实现时，需要存储两个矩阵：话题-单词矩阵 $N_{K \times V} = [n_{kv}]$ 和文本-话题矩阵 $N_{M \times K} = [n_{mk}]$。每次在抽样前，需要先将对应位置的话题数减 1，再进行抽样，抽样后按抽样结果在对应位置将话题数加 1。潜在狄利克雷分配的吉布斯抽样算法如下：

输入：单词总集合为 $w = (w_{11}, w_{12}, ..., w_{1N_1}, w_{21}, w_{22}, ..., w_{2N_2}, ..., w_{M1}, w_{M2}, ..., w_{MN_M})$，超参数 α, β 和话题数 K。

1. 将计数矩阵的元素 n_{mk}, n_{kv}，计数向量 n_m, n_k 初值置为 0
2. 对所有单词 $w_{mn}, m = 1, 2, ..., M; n = 1, 2, ..., N_m$ 进行以下操作

- 抽样话题 $z_{mn} = z_k \sim Multi(\frac{1}{K})$
- 将计数矩阵和计数向量中的对应元素 n_{mk}, n_{kv}, n_m, n_k 加 1

3. 对所有单词 $w_{mn}, m = 1, 2, ..., M; n = 1, 2, ..., N_m$ 进行以下操作，直至进入燃烧期

- 当前的单词 w_{mn} 是第 v 个单词，话题 z_{mn} 是第 k 个话题
- 将计数矩阵和计数向量中的对应元素 n_{mk}, n_{kv}, n_m, n_k 减 1
- 按满条件分布抽样

$$p(z_i|z_{-i}, w, \alpha, \beta) \propto \frac{n_{kv} + \beta_v}{\sum_{v=1}^{V}(n_{kv} + \beta_v)} \frac{n_{mk} + \alpha_k}{\sum_{k=1}^{K}(n_{mk} + \alpha_k)}$$

- 得到新话题 k'，分配给 z_{mn}。将计数矩阵和计数向量中的对应元素 $n_{mk'}, n_{k'v}, n_m, n_{k'}$ 加 1

4. 根据计数矩阵 $N_{K\times V}=[n_{kv}]$ 和 $N_{M\times K}=[n_{mk}]$ 计算参数的值

$$\theta_{mk}=\frac{n_{mk}+\alpha_k}{\sum_{k=1}^{K}(n_{mk}+\alpha_k)}, m=1,2,...,M; k=1,2,...,K$$

$$\phi_{kv}=\frac{n_{kv}+\beta_v}{\sum_{v=1}^{V}(n_{kv}+\beta_v)}, k=1,2,...,K; v=1,2,...,V$$

输出：模型的参数 θ, ϕ。

1.4 LDA 的变分 EM 算法

变分推理的目标是学习模型的后验概率分布 $f(z|x)$。其使用一个概率分布 $q(z)$ 去近似复杂的概率分布 $f(z|x)$。使用 KL 散度 $D(q(z)||f(z|x))$ 计算两个分布之间的相似度，从中找出 KL 散度最小的变分分布 $q^*(z)$ 去近似 $f(z|x)$，即 $q^*(z)\approx f(z|x)$。KL 散度可以写成如下形式：

$$\begin{aligned}D(q(z)||f(z|x)) &= E_q[\log q(z)]-E_q[\log f(z|x)]\\ &= E_q[\log q(z)]-E_q[\log f(x,z)]+E_q[\log p(x)]\\ &= \log p(x)-\{E_q[\log f(x,z)]-E_q[\log q(z)]\}\end{aligned}$$

注意到 KL 散度大于等于 0，因此有

$$\log p(x)\geq E_q[\log f(x,z)]-E_q[\log q(z)]$$

不等式左端称为证据，右端称为证据下界。将证据下界记为 $L(q)=E_q[\log f(x,z)]-E_q[\log q(z)]$。KL 散度的最小化等价于证据下界 $L(q)$ 的最大化。

此外，为了防止变分分布 $q(z)$ 的搜索范围过大，致使出现不可计算问题。变分分布 $q(z)$ 需要定义在**平均场**上，其对 z 的所有变量都是相互独立的，即 $q(z)=q(z_1)q(z_2)...q(z_n)$。因此，变分推理主要有以下几个步骤：

1. 定义变分分布 $q(z)$
2. 推导其证据下界表达式
3. 使用最优化方法对证据下界进行优化，得到最优分布 $q^*(z)$，作为 $p(z|x)$ 的近似

其中，最优化方法可以选用 EM 算法，得到**变分 EM 算法**。假设模型的概率分布是 $p(x,z|\theta), x$ 是观测变量，z 是隐变量，θ 是参数。导入平均场 $q(z)=\prod_{i=1}^{n}q(z_i)$，则可定义证据下界

$$L(q,\theta)=E_q[\log p(x,z|\theta)]-E_q[\log q(z)]$$

变分 EM 算法分别对 q,θ 进行迭代以求证据下界的最大值，其步骤如下：

1. E 步：固定 θ，求 $L(q,\theta)$ 关于 q 的最大化
2. M 步：固定 q，求 $L(q,\theta)$ 关于 θ 的最大化

将变分 EM 算法用于 LDA 模型中时，可以如下引入证据下界。简单起见，每次只考虑一个文本。文本的单词序列为 $w=(w_1,w_2,...,w_N)$，对应的话题序列为 $z=(z_1,z_2,...,z_N)$，话题分布的参数为 θ，其联合分布为：

$$p(\theta,w,z|\alpha,\phi)=p(\theta|\alpha)\prod_{n=1}^{N}p(z_n|\theta)p(w_n|z_n,\phi)$$

由于 θ,z 是隐变量，可定义平均场 $q(\theta,z|\gamma,\eta)=q(\theta|\gamma)\prod_{n=1}^{N}q(z_n|\eta_n)$。其中 γ 是 θ 服从的狄利克雷分布的参数，$\eta=(\eta_1,\eta_2,...,\eta_N)$ 是 $z_1,z_2,...,z_N$ 服从的多项分布的参数。因此，文本的证据下界为

$$L(\gamma,\eta,\alpha,\phi)=E_q[\log p(\theta,w,z|\alpha,\phi)]-E_q[\log q(\theta,z|\gamma,\eta)]$$

其中，γ,η 是变分分布的参数，α,ϕ 是 LDA 模型的参数。通过对函数 $L(\gamma,\eta,\alpha,\phi)$ 最大化即可求得变分分布的参数和 LDA 模型的参数。变分分布的参数优化方法是

$$\eta_{nk}\propto\phi_{kv}\exp\left(\Psi(\gamma_k)-\Psi\left(\sum_{l=1}^{K}\gamma_l\right)\right)$$

$$\gamma_k=\alpha_k+\sum_{n=1}^{N}\eta_{nk}$$

其中 $\Psi(.)$ 为对数伽马函数的导函数，即 $\frac{\partial}{\partial x}\log\Gamma(x)=\Psi(x)$。在得到 η_{nk} 后，需要做放缩使得 $\sum_{k=1}^{K}\eta_{nk}=1$。

模型参数的更新方法是

$$\phi_{kv}=\sum_{m=1}^{M}\sum_{n=1}^{N_m}\eta_{mnk}w_{mn}^{v}$$

其中，η_{mnk} 表示第 m 个文本的第 n 个单词属于第 k 个话题的概率；w_{mn}^{v} 当第 m 个文本的第 n 个单词是单词集合中的第 v 个单词时为 1，否则为 0。

α 的更新需要通过牛顿法得到 $\alpha:=\alpha-H^{-1}(\alpha)g(\alpha)$。其中，$g(\alpha)$ 为一阶导，$H(\alpha)$ 为黑塞矩阵，其元素参数如下：

$$\frac{\partial L}{\partial\alpha_k}=M\left[\Psi\left(\sum_{l=1}^{K}\alpha_l\right)-\Psi(\alpha_k)\right]+\sum_{m=1}^{M}\left[\Psi(\gamma_{mk})-\Psi\left(\sum_{l=1}^{K}\gamma_{ml}\right)\right]$$

$$\frac{\partial^2 L}{\partial\alpha_k\partial\alpha_l}=M\left[\Psi'\left(\sum_{l=1}^{K}\alpha_l\right)-I(k=l)\Psi'(\alpha_k)\right]$$

因此，LDA 的变分 EM 算法步骤如下：

输入：单词总集合为 $w = (w_{11}, w_{12}, ..., w_{1N_1}, w_{21}, w_{22}, ..., w_{2N_2}, ..., w_{M1}, w_{M2}, ..., w_{MN_M})$ 和话题个数 K。

1. 初始化变分参数 γ, η 和模型参数 α, ϕ
2. 固定模型参数 α, ϕ，更新变分参数 γ, η

- 更新 $\eta_{nk} := \phi_{kv} \exp\left(\Psi(\gamma_k) - \Psi\left(\sum_{l=1}^{K} \gamma_l\right)\right)$
- 规范化使得 $\sum_{k=1}^{K} \eta_{nk} = 1$
- 更新 $\gamma = \alpha + \sum_{n=1}^{N} \eta_n$
- 重复以上直至收敛

3. 固定变分参数 γ, η，更新模型参数 α, ϕ

- 更新 $\phi_{kv} = \sum_{m=1}^{M} \sum_{n=1}^{N_m} \eta_{mnk} w_{mn}^{v}$
- 更新 $\alpha := \alpha - H^{-1}(\alpha) g(\alpha)$
- 重复以上直至收敛

输出：变分分布的参数 γ, η 和 LDA 模型的参数 α, ϕ。

2 代码实现

本次使用模拟的数据集。数据集中共 100 个文本，每个文本中共 20 个单词，单词集合中共 12 个单词。

```
from sklearn.datasets import make_multilabel_classification
X, _ = make_multilabel_classification(random_state=0)
X
```

```
## array([[3., 1., 4., ..., 4., 1., 3.],
##        [5., 0., 6., ..., 0., 0., 3.],
##        [3., 4., 1., ..., 3., 2., 5.],
##        ...,
##        [2., 1., 2., ..., 1., 0., 3.],
##        [6., 4., 1., ..., 1., 3., 5.],
##        [2., 4., 2., ..., 5., 4., 2.]])
```

2.1 sklearn 实现

潜在狄利克雷分配的 sklearn 接口位于 sklearn.decomposition.LatentDirichletAllocation, 其文档可见**此处**。其中较为重要的参数有：

- n_components：话题个数 K

- doc_topic_prior：α
- topic_word_prior：β

注意，在 sklearn 中，LDA 是使用变分 EM 算法进行参数估计的。

```
from sklearn.decomposition import LatentDirichletAllocation
clf = LatentDirichletAllocation(n_components=5)
clf.fit(X)
```

后两个文本的话题概率分布为：

```
print(clf.transform(X[-2:,:]))
```

```
## [[0.14655242 0.12308133 0.00360182 0.72315697 0.00360747]
##  [0.31658403 0.00360047 0.10941854 0.0039823  0.56641466]]
```

2.2 吉布斯抽样算法

使用如下类实现吉布斯抽样算法的 LDA 模型参数估计：

```
import numpy as np

class GibbsLDA:

    def __init__(self,K,alpha=None,beta=None,sim_time=1000):
        self.K = K
        self.alpha = alpha
        self.beta = beta
        self.sim_time = sim_time

    def fit(self,X):
        M,N = X.shape
        V = len(np.unique(X.reshape(-1)))
        K = self.K

        N_kv = np.zeros((K,V))
        N_mk = np.zeros((M,K))
        if self.alpha is None:
            alpha = [1 for _ in range(K)]
        else:
            alpha = self.alpha
```

```
        if self.beta is None:
            beta = [1 for _ in range(V)]
        else:
            beta = self.beta

        topic_mat = np.zeros((M,N))
        for m in range(M):
            for n in range(N):
                word = X[m,n]
                topic = int(K*np.random.random())
                topic_mat[m,n] = topic
                N_mk[m,topic] += 1
                N_kv[topic,int(word)] +=1

        for _ in range(self.sim_time):
            for m in range(M):
                for n in range(N):
                    word = X[m,n]
                    topic = int(topic_mat[m,n])
                    N_mk[m,topic] -= 1
                    N_kv[topic,int(word)] -= 1

                    prob = (N_kv[:,int(word)]+beta[int(word)])/(np.sum(N_kv+beta,axis=1))\
                          *(N_mk[m,:]+alpha[topic])
                    prob = list(prob/np.sum(prob))
                    topic = list(np.random.multinomial(1,prob)).index(1)

                    topic_mat[m,n] = topic
                    N_mk[m,topic] += 1
                    N_kv[topic,int(word)] += 1

        theta = N_mk+alpha
        self.theta = theta/np.sum(theta,1).reshape(-1,1)
        phi = N_kv+beta
        self.phi = phi/np.sum(phi,1).reshape(-1,1)


    def transform(self):
```

```
        return self.theta
```

后两个文本的话题概率分布为，此处仅优化 100 次：

```
clf = GibbsLDA(K=5,sim_time=100)
clf.fit(X)
print(clf.transform()[-2:])
```

```
## [[0.04 0.16 0.2  0.28 0.32]
##  [0.24 0.36 0.04 0.28 0.08]]
```

PageRank

1 PageRank

1.1 模型定义

PageRank 算法是计算互联网网页重要程度的算法，它对每一个网页给出一个正实数（即 PageRank 值），表示网页的重要程度。PageRank 值越高，网页就越重要。PageRank 值依赖网络的拓扑结构，当网络结构确定后，PageRank 值就确定。PageRank 基本定义是在一个**有向图**上的**随机游走模型**。

定义（PageRank 的基本定义）：给定一个包含 n 个节点 $v_1, v_2, ..., v_n$ 的强连通且非周期性的有向图，在有向图上定义随机游走模型（相当于一阶马尔可夫链）。随机游走的特点是从一个节点到有有向边连出的所有节点的转移概率相等，转移矩阵为 M。该马尔可夫链具有平稳分布R，满足 $MR = R$。平稳分布 R 称为这个有向图的 PageRank。

记 $R = [PR(v_1), PR(v_2), ..., PR(v_n)]^T$，易于发现：

$$PR(v_i) \geq 0, \quad i = 1, 2, ..., n$$

$$\sum_{i=1}^{n} PR(v_i) = 1$$

$$PR(v_i) = \sum_{v_j \in M(v_i)} \frac{PR(v_j)}{L(v_j)}, \quad i = 1, 2, ..., n$$

其中，$M(v_i)$ 表示指向 v_i 的节点集合，$L(v_j)$ 表示节点 v_j 连出的有向边的个数。

按以上定义方法定义的 PageRank 必须满足**强连通和非周期性的条件**。但实际情况中，大部分有向图可能存在孤立点等情况，不满足这一条件，致使其平稳分布也不存在。可使用**平滑方法**解决这一问题，可假设存在另一个完全随机游走，其转移矩阵的元素全部为 $\frac{1}{n}$。将两个转移矩阵的线性组合构成一个新的转移矩阵，按新的转移矩阵定义一个新的马尔可夫链。该马尔可夫链的转移矩阵为 $M' = dM + \frac{1-d}{n}1_{n\times n}$，其中 $0 \leq d \leq 1$ 是阻尼因子，$1_{n\times n}$ 是元素均为 1 的 $n \times n$ 的矩阵。可以证明，该马尔可夫链一定具有平稳分布，且平稳分布 R 满足

$$R = dMR + \frac{1-d}{n}1_n$$

R 即为有向图的一般 PageRank，同样可以得到：

$$PR(v_i) > 0, \quad i = 1, 2, ..., n$$

$$\sum_{i=1}^{n} PR(v_i) = 1$$

$$PR(v_i) = d\left(\sum_{v_j \in M(v_i)} \frac{PR(v_j)}{L(v_j)}\right) + \frac{1-d}{n}, \quad i = 1, 2, ..., n$$

注意，由于加入了平滑项，因此所有节点的 PageRank 值都不会为 0。

1.2 PageRank 的计算

PageRank 的计算主要有代数算法，迭代算法和幂法。

1.2.1 代数算法

由于 $R = dMR + \frac{1-d}{n}1_n$，于是

$$(I - dM)R = \frac{1-d}{n}1_n$$

$$R = (I - dM)^{-1}\frac{1-d}{n}1_n$$

当 $0 < d < 1$ 时，代数算法可以求得唯一解。但对于节点数较多的网络，计算逆矩阵的计算量极大，因此该方法不常使用。

1.2.2 迭代算法

迭代算法使用递推式 $R_{t+1} = dMR_t + \frac{1-d}{n}1_n$ 进行迭代，直至收敛。其算法如下：

输入：含有 n 个节点的有向图，转移矩阵 M，阻尼因子 d。

1. 初始化向量 R_0
2. 不断计算 $R_{t+1} = dMR_t + \frac{1-d}{n}1_n$
3. 如果 R_{t+1} 和 R_t 充分接近，停止迭代，令 $R = R_{t+1}$；否则回到 2

输出：有向图的 PageRank 向量 R。

1.2.3 幂法

Perron-Frobenius 定理证明了一般 PageRank 向量 R 是转移矩阵 $A = dM + \frac{1-d}{n}I_{n\times n}$ 的主特征向量。幂法通过近似计算矩阵的主特征值和主特征向量求得有向图的一般 PageRank 值。

n 阶矩阵 A 的主特征值和主特征向量可采用如下的步骤求得。首先，任取一个 n 维向量 x_0，构造如下序列

$$x_0, \quad x_1 = Ax_0, \quad x_2 = Ax_1, \quad ..., \quad x_k = Ax_{k-1}$$

记 A 的 n 个特征值按绝对值大小排列有 $|\lambda_1| \geq |\lambda_2| \geq ... \geq |\lambda_n|$，对应的特征向量为 $u_1, u_2, ..., u_n$。将初始向量 x_0 表示为 $u_1, u_2, ..., u_n$ 的线性组合：$x_0 = a_1u_1 + a_2u_2 + ... + a_nu_n$。因此有

$$\begin{aligned} x_k = A^kx_0 &= a_1A^ku_1 + a_2A^ku_2 + ... + a_nA^ku_k \\ &= a_1\lambda_1^ku_1 + a_2\lambda_2^ku_2 + ... + a_n\lambda_n^ku_n \\ &= a_1\lambda_1^k\left[u_1 + \frac{a_2}{a_1}\left(\frac{\lambda_2}{\lambda_1}\right)^k u_2 + ... + \frac{a_n}{a_1}\left(\frac{\lambda_n}{\lambda_1}\right)^k u_n\right] \end{aligned}$$

当 $k \to +\infty, x_k \to a_1\lambda_1^ku_1$，因此

$$\begin{aligned} x_k &\approx a_1\lambda_1^ku_1 \\ x_{k+1} &\approx a_1\lambda_1^{k+1}u_1 \end{aligned}$$

因此，$\lambda_1 \approx \frac{x_{k+1,j}}{x_{k,j}}$。此外，在每一次的迭代过程中，需要进行规范化操作，即：

$$\begin{aligned} y_{t+1} &= Ax_t \\ x_{t+1} &= \frac{y_{t+1}}{||y_{t+1}||_\infty} \end{aligned}$$

其中，$||y_{t+1}||_\infty = \max\{|x_1|, |x_2|, ..., |x_n|\}$。因此，PageRank 的幂法算法如下：

输入：含有 n 个节点的有向图，转移矩阵 M，阻尼因子 d。

1. 初始化向量 x_0
2. 计算转移矩阵 $A = dM + \frac{1-d}{n}I_{n\times n}$
3. 计算并规范化

$$\begin{aligned} y_{t+1} &= Ax_t \\ x_{t+1} &= \frac{y_{t+1}}{||y_{t+1}||_\infty} \end{aligned}$$

4. 当 $||x_{t+1} - x_t|| < \epsilon$ 时停止迭代，令 $R = x_{t+1}$；否则，回到 3
5. 对 R 进行规范化操作，使其变为概率分布

输出：有向图的 PageRank 向量 R。

2 代码实现

此次考虑的有向图的转移矩阵为：

$$M = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 1 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

PageRank 算法在 sklearn 中没有接口，但在 scikit-network 中有接口。此处我们不调用接口，直接对迭代算法和幂法进行实现。

准备数据：

```
import numpy as np
M = np.array([[0,1/2,0,0],
              [1/3,0,0,1/2],
              [1/3,0,1,1/2],
              [1/3,1/2,0,0]])
```

2.1 迭代算法

```
class IterPageRank:

    def __init__(self,d=0.8,init=None):
        self.d = d
        self.init = init

    def fit(self,M):
        n = M.shape[0]
        if self.init is None:
            r = np.ones((n,1))/n
        else:
```

```
            r = self.init

        err = 1
        while err>1e-3:
            r_new = self.d*M.dot(r)+(1-self.d)/n*np.ones((n,1))
            err = np.mean(np.abs(r_new-r))
            r = r_new

        self.pr = r
```

结果为：

```
clf = IterPageRank()
clf.fit(M)
print(clf.pr)
```

```
## [[0.10180032]
##  [0.12903271]
##  [0.64013426]
##  [0.12903271]]
```

2.2 幂法

```
class PowerPageRank:

    def __init__(self,d=0.8,init=None):
        self.d = d
        self.init = init

    def fit(self,M):
        n = M.shape[0]
        if self.init is None:
            r = np.ones((n,1))/n
        else:
            r = self.init
        A = self.d*M+(1-self.d)/n*np.ones((n,1))

        err = 1
        while err>1e-3:
```

```
            r_new = A.dot(r)
            r_new = r_new/np.max(r_new)
            err = np.mean(np.abs(r_new-r))
            r = r_new

        self.pr = r/np.sum(r)
```

结果为：

```
clf = PowerPageRank()
clf.fit(M)
print(clf.pr)
```

```
## [[0.10180032]
##  [0.12903271]
##  [0.64013426]
##  [0.12903271]]
```