第 3 章

CHAPTER 3

Python 数据降维

伴随 ICT(通信与信息技术)和互联网技术的不断发展,人们收集和获得数据的能力越来越强。而这些数据已呈现出维数高、规模大和结构复杂等特点。

人们想利用这些大数据(维数大、规模大、复杂大),挖掘其中有意义的知识和内容以指导实际生产和具体应用,数据的降维就显得尤为重要了。数据降维又称为维数约简。顾名思义,就是降低数据的维数。为什么要降低数据的维数?如何有效地降低数据的维数?由此问题引发了广泛的研究和应用。

数据降维,一方面可以解决“维数灾难”,缓解“信息丰富、知识贫乏”现状,降低复杂度;另一方面可以更好地认识和理解数据。

截止到目前,数据降维的方法很多。从不同的角度入手可以有着不同的分类,主要分类方法有:根据数据的特性可以划分为线性降维和非线性降维,根据是否考虑和利用数据的监督信息可以划分为无监督降维、有监督降维和半监督降维,根据保持数据的结构可以划分为全局保持降维、局部保持降维和全局与局部保持一致降维等。

总之,数据降维意义重大,数据降维方法众多,很多时候需要根据特定问题选用合适的数据降维方法。数据降维是机器学习领域中非常重要的内容。

3.1 维度灾难与降维

1. 维度灾难

维度灾难(curse of dimensionality)用来描述当(数学)空间维度增加时,分析和组织高维空间(通常有成百上千维),因体积指数增加而遇到各种问题场景。在机器学习中,维度灾难常指以下问题:

在高维情况下,数据样本稀疏。

例如,k 近邻法的讨论中经常涉及维度灾难,是因为 k 近邻法基于一个重要的基本假设:任意样本附近任意小的距离内总能找到一个训练样本,即训练样本的采样密度足够大,也称为“密采样”,才能保证分类性能;当特征维度很大时,满足密采样的样本数量会呈指数级增长,大到几乎无法达到。

在高维情况下,涉及距离、内积的计算变得困难。

其实,不仅是 k 近邻,其他机器学习算法几乎都会遇到维度灾难的问题。

2. 降维

缓解维度灾难的一个重要途径就是降维。

1) 为什么能够进行降维

这是因为很多时候,数据是高维的,但是与学习任务(分类、回归等)密切相关的仅是某个低维分布,即高维空间中的某个低维难嵌入。因此,很多情况下,高维空间中的样本点,在低维嵌入子空间中更容易学习。

2) 线性降维

一般来说,想获得低维子空间,最简单的方法是对原始高维空间进行线性变换:

给定 d 维空间中的样本 $\boldsymbol{X}=(x_1,x_2,\cdots,x_m)\in\mathbf{R}^{d\times m}$,变换之后得到 $d'\leqslant d$ 维空间中的样本

$$\boldsymbol{Z}=\boldsymbol{W}^{\mathrm{T}}\boldsymbol{X}$$

其中,$\boldsymbol{W}\in\mathbf{R}^{d\times d'}$ 是变换矩阵,$\boldsymbol{Z}\in\mathbf{R}^{d'\times m}$ 是样本在新空间中的表达。

变换矩阵 $\boldsymbol{W}$ 可视为 d' 个 d 维基向量,新空间中的属性是原空间属性的线性组合,基于线性变换来进行降维的方法称为线性维方法,都符合上面的式子,主要区别在于对低维子空间的性质有所不同,相当于对 $\boldsymbol{W}$ 施加了不同的约束。

3) 降维效果的评估

通常通过比较降维前后学习器性能,若性能有提高,则认为降维起到了作用。针对已经降到二维或者三维的情况,可以利用可视化技术直观地判断降维效果。

3.2 主成分分析

主成分分析(Principal Component Analysis,PCA)是一种最常用的无监督降维方法,通过降维技术把多个变量化为少数几个主成分(综合变量)的统计分析方法。这些主成分能够反映原始变量的绝大部分信息,它们通常表示为原始变量的某种线性组合。

3.2.1 PCA 原理

为了便于维度变换,有如下假设。

- 假设样本数据是 n 维的。
- 假设原始坐标系为:由标准正交基向量$\{\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n\}$张成的空间,其中 $\|\vec{i}_s\|=1$;$\vec{i}_s\cdot\vec{i}_t=0,s\neq t$。
- 假设经过线性变换后的新坐标系为:由标准正交基向量$\{\vec{j}_1,\vec{j}_2,\cdots,\vec{j}_n\}$张成的空间,其中 $\|\vec{j}_s\|=1$;$\vec{j}_s\cdot\vec{j}_t=0,s\neq t$。

根据定义,有:

$$\vec{j}_s=(\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n)\begin{bmatrix}\vec{j}_s\cdot\vec{i}_1\\ \vdots\\ \vec{j}_s\cdot\vec{i}_n\end{bmatrix},s=1,2,\cdots,n$$

记 $\boldsymbol{w}_s=(\vec{j}_s\cdot\vec{i}_1,\vec{j}_s\cdot\vec{i}_2,\cdots,\vec{j}_s\cdot\vec{i}_n)^{\mathrm{T}}$（它是一个列向量，但是为了与基向量做区分，这里没有给出向量的箭头符号），其各分量就是基向量 $\vec{j}_s$ 在原始坐标系 $\{\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n\}$ 中的投影。即：$\vec{j}_s=(\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n)\cdot\boldsymbol{w}_s$。根据标准正交基的性质，有：

- $\|\boldsymbol{w}_s\|=1,s=1,2,\cdots,n$；
- $\boldsymbol{w}_s\cdot\boldsymbol{w}_t=0,s\neq t$。

根据定义，有：

$$(\vec{j}_1,\vec{j}_2,\cdots,\vec{j}_n)=(\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n)(\boldsymbol{w}_1,\boldsymbol{w}_2,\cdots,\boldsymbol{w}_n)$$

令坐标变换矩阵 $\boldsymbol{W}$ 为：

$$\boldsymbol{W}=(\boldsymbol{w}_1,\boldsymbol{w}_2,\cdots,\boldsymbol{w}_n)=\begin{bmatrix}\vec{j}_1\cdot\vec{i}_1 & \vec{j}_2\cdot\vec{i}_1 & \cdots & \vec{j}_n\cdot\vec{i}_1\\ \vec{j}_1\cdot\vec{i}_2 & \vec{j}_2\cdot\vec{i}_2 & \cdots & \vec{j}_n\cdot\vec{i}_2\\ \vdots & \vdots & \ddots & \vdots\\ \vec{j}_1\cdot\vec{i}_n & \vec{j}_2\cdot\vec{i}_n & \cdots & \vec{j}_n\cdot\vec{i}_n\end{bmatrix}$$

则有

$$(\vec{j}_1,\vec{j}_2,\cdots,\vec{j}_n)=(\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n)\boldsymbol{W}$$

$\boldsymbol{W}$ 的第 s 列就是 $\vec{j}_s$ 在原始坐标系 $\{\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n\}$ 中的投影，且有 $\boldsymbol{W}=\boldsymbol{W}^{\mathrm{T}}$，$\boldsymbol{W}\boldsymbol{W}^{\mathrm{T}}=\boldsymbol{I}$（即它的逆矩阵就是它的转置）。

假设样本点 $\vec{x}_i$ 在原始坐标系中的表示为：

$$\vec{x}_i=(\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n)\begin{bmatrix}x_i^{(1)}\\ x_i^{(2)}\\ \vdots\\ x_i^{(n)}\end{bmatrix}$$

令 $\boldsymbol{x}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})^{\mathrm{T}}$，则 $\vec{x}_i=(\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n)\boldsymbol{x}_i$。

假设样本点 $\vec{x}_i$ 在新坐标系中的表示为：

$$\vec{x}_i=(\vec{j}_1,\vec{j}_2,\cdots,\vec{j}_n)\begin{bmatrix}z_i^{(1)}\\ z_i^{(2)}\\ \vdots\\ z_i^{(n)}\end{bmatrix}$$

令 $\boldsymbol{z}_i=(z_i^{(1)},z_i^{(2)},\cdots,z_i^{(n)})^{\mathrm{T}}$，则 $\vec{x}_i=(\vec{j}_1,\vec{j}_2,\cdots,\vec{j}_n)\boldsymbol{z}_i$。根据 $\vec{x}_i=\vec{x}_i$，则有：

$$(\vec{j}_1,\vec{j}_2,\cdots,\vec{j}_n)\boldsymbol{z}_i=(\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n)\boldsymbol{W}\boldsymbol{z}_i=(\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n)\boldsymbol{x}_i$$

于是有：

$$\boldsymbol{z}_i=\boldsymbol{W}^{-1}\boldsymbol{x}_i=\boldsymbol{W}^{\mathrm{T}}\boldsymbol{x}_i$$

丢弃其中的部分坐标，将维度降低到 $d<n$，则样本点 $\vec{x}_i$ 在低维坐标系中的坐标为 $\boldsymbol{z}_i'=(z_i^{(1)},z_i^{(2)},\cdots,z_i^{(n)})^{\mathrm{T}}$。现在的问题是：最好丢弃哪些坐标？想法是：基于降低之后的坐标重构样本时，尽量要与原始样本相近。

如果基于降维后的坐标 $\boldsymbol{z}_i'$ 来重构 $\vec{x}_i$：

$$\hat{\vec{x}}_i=(\vec{j}_1,\vec{j}_2,\cdots,\vec{j}_d)\begin{bmatrix}z_i^{(1)}\\z_i^{(2)}\\\vdots\\z_i^{(d)}\end{bmatrix}=(\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n)(\boldsymbol{w}_1,\boldsymbol{w}_2,\cdots,\boldsymbol{w}_d)\begin{bmatrix}z_i^{(1)}\\z_i^{(2)}\\\vdots\\z_i^{(d)}\end{bmatrix}$$

$$=(\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n)(\boldsymbol{w}_1,\boldsymbol{w}_2,\cdots,\boldsymbol{w}_d)\begin{bmatrix}\boldsymbol{w}_1^{\mathrm{T}}\cdot\boldsymbol{x}_i\\\boldsymbol{w}_2^{\mathrm{T}}\cdot\boldsymbol{x}_i\\\vdots\\\boldsymbol{w}_d^{\mathrm{T}}\cdot\boldsymbol{x}_i\end{bmatrix}$$

$$=(\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n)(\boldsymbol{w}_1,\boldsymbol{w}_2,\cdots,\boldsymbol{w}_d)\begin{bmatrix}\boldsymbol{w}_1^{\mathrm{T}}\\\boldsymbol{w}_2^{\mathrm{T}}\\\vdots\\\boldsymbol{w}_d^{\mathrm{T}}\end{bmatrix}\cdot\boldsymbol{x}_i$$

令 $\boldsymbol{W}_d=(\boldsymbol{w}_1,\boldsymbol{w}_2,\cdots,\boldsymbol{w}_d)$，即它是坐标变换矩阵 $\boldsymbol{W}$ 的前 d 列，则：

$$\hat{\vec{x}}_i=(\vec{i}_1,\vec{i}_2,\cdots,\vec{i}_n)\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{x}_i$$

考虑整个训练集，原样本点 $\vec{x}_i$ 和基于投影重构的样本点 $\hat{\vec{x}}_i$ 之间的距离为（即所有重构的样本点与原样本点的整体误差）：

$$\sum_{i=1}^{N}\|\hat{\vec{x}}_i-\vec{x}_i\|_2^2=\sum_{i=1}^{N}\|x_i-\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{x}_i\|_2^2$$

考虑：

$$\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{x}_i=(\boldsymbol{w}_1,\boldsymbol{w}_2,\cdots,\boldsymbol{w}_d)\begin{bmatrix}\boldsymbol{w}_1^{\mathrm{T}}\\\boldsymbol{w}_2^{\mathrm{T}}\\\vdots\\\boldsymbol{w}_d^{\mathrm{T}}\end{bmatrix}\boldsymbol{x}_i=\sum_{s=1}^{d}\boldsymbol{w}_s(\boldsymbol{w}_s^{\mathrm{T}}\boldsymbol{x}_i)$$

由于 $\boldsymbol{w}_s^{\mathrm{T}}\boldsymbol{x}_i$ 是标量，所以有：

$$\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{x}_i=\sum_{s=1}^{d}(\boldsymbol{w}_s^{\mathrm{T}}\boldsymbol{x}_i)\boldsymbol{w}_s$$

由于 $\boldsymbol{w}_s^{\mathrm{T}}\boldsymbol{x}_i$ 是标量，所以它的转置等于它本身，所以有：

$$\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{x}_i=\sum_{s=1}^{d}(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}_s)\boldsymbol{w}_s$$

于是有：

$$\sum_{i=1}^{N}\|\hat{\vec{x}}_i-\vec{x}_i\|_2^2=\sum_{i=1}^{N}\|\boldsymbol{x}_i-\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{x}_i\|_2^2=\sum_{i=1}^{N}\|\boldsymbol{x}_i-\sum_{s=1}^{d}(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}_s)\boldsymbol{w}_s\|_2^2$$

定义矩阵 $\boldsymbol{X}=(\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_N)$，即矩阵 $\boldsymbol{X}$ 的第 i 列就是 $\boldsymbol{x}_i$。即可以证明：

$$\|\boldsymbol{X}^{\mathrm{T}}-\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\|_{\mathrm{F}}^2=\sum_{i=1}^{N}\|x_i-\sum_{s=1}^{d}(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}_s)\boldsymbol{w}_s\|_2^2$$

其中，$\|\cdot\|_{\mathrm{F}}$ 为矩阵的 Frobenius 范数（简称 F 范数）。接下来的证明过程中，要用到矩阵的 F 范数和矩阵的迹的性质：

(1) 矩阵 $\boldsymbol{A}$ 的 F 范数定义为：$\|\boldsymbol{A}\|_{\mathrm{F}}=\sqrt{\sum_i\sum_j a_{ij}^2}$，即矩阵所有元素的平方和的开方。F 范数的性质有：

- $\|\boldsymbol{A}\|_{\mathrm{F}}=\|\boldsymbol{A}^{\mathrm{T}}\|_{\mathrm{F}}$。
- $\|\boldsymbol{A}\|_{\mathrm{F}}=\mathrm{tr}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})$，tr 为矩阵的迹。

(2) 对于方阵，矩阵的迹定义为：$\mathrm{tr}(\boldsymbol{A})=\sum_i a_{ii}$，即矩阵对角线元素之和。矩阵的迹的性质有：

- $\mathrm{tr}(\boldsymbol{A})=\mathrm{tr}(\boldsymbol{A}^{\mathrm{T}})$。
- $\mathrm{tr}(\boldsymbol{A}\pm\boldsymbol{B})=\mathrm{tr}(\boldsymbol{A})\pm\mathrm{tr}(\boldsymbol{B})$。
- 如果 $\boldsymbol{A}$ 为 $m\times n$ 阶矩阵，$\boldsymbol{B}$ 为 $n\times m$ 阶矩阵，则 $\mathrm{tr}(\boldsymbol{AB})=\mathrm{tr}(\boldsymbol{BA})$。
- 矩阵的迹等于矩阵的特征值之和：$\mathrm{tr}(\boldsymbol{A})=\lambda_1+\lambda_2+\cdots+\lambda_n$。
- 对任何正整数 k 有：$\mathrm{tr}(\boldsymbol{A}^k)=\lambda_1^k+\lambda_2^k+\cdots+\lambda_n^k$。

要求解最优化问题：

$$\begin{aligned}\boldsymbol{W}_d^* &=\underset{\boldsymbol{W}_d}{\mathrm{argmin}}\|\hat{\vec{x}}_i-\vec{x}_i\|_2^2\\&=\underset{\boldsymbol{W}_d}{\mathrm{argmin}}\|\boldsymbol{X}^{\mathrm{T}}-\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\|_{\mathrm{F}}^2\\&=\underset{\boldsymbol{W}_d}{\mathrm{argmin}}\mathrm{tr}[(\boldsymbol{X}^{\mathrm{T}}-\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})^{\mathrm{T}}(\boldsymbol{X}^{\mathrm{T}}-\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})]\\&=\underset{\boldsymbol{W}_d}{\mathrm{argmin}}\mathrm{tr}[(\boldsymbol{X}-\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{X})(\boldsymbol{X}^{\mathrm{T}}-\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})]\\&=\underset{\boldsymbol{W}_d}{\mathrm{argmin}}\mathrm{tr}[\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}-\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}-\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}+\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}]\\&=\underset{\boldsymbol{W}_d}{\mathrm{argmin}}\mathrm{tr}[\mathrm{tr}(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}})-\mathrm{tr}(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})-\mathrm{tr}(\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}})+\mathrm{tr}(\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})]\end{aligned}$$

因为矩阵及其转置的迹相等，因此 $\mathrm{tr}(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})=\mathrm{tr}(\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}})$。由于可以在 tr(•)中调整矩阵的顺序，则 $\mathrm{tr}(\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})=\mathrm{tr}(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})$。

考虑到：

$$\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}=\begin{bmatrix}\boldsymbol{w}_1^{\mathrm{T}}\\\boldsymbol{w}_2^{\mathrm{T}}\\\vdots\\\boldsymbol{w}_d^{\mathrm{T}}\end{bmatrix}(\boldsymbol{w}_1,\boldsymbol{w}_2,\cdots,\boldsymbol{w}_d)=\boldsymbol{I}_{d\times d}$$

代入上式有：

$$\mathrm{tr}(\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}}\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})=\mathrm{tr}(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})$$

于是：

$$\begin{aligned}\boldsymbol{W}_d^* &=\underset{\boldsymbol{W}_d}{\mathrm{argmin}}[\mathrm{tr}(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}})-2\mathrm{tr}(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})+\mathrm{tr}(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})]\\&=\underset{\boldsymbol{W}_d}{\mathrm{argmin}}[\mathrm{tr}(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}})-\mathrm{tr}(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})]\end{aligned}$$

由于 $\mathrm{tr}(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}})$ 与 $\boldsymbol{W}_d$ 无关，因此，

$$\boldsymbol{W}_d^*=\underset{\boldsymbol{W}_d}{\mathrm{argmin}}-\mathrm{tr}(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})=\underset{\boldsymbol{W}_d}{\mathrm{argmin}}\mathrm{tr}(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}_d\boldsymbol{W}_d^{\mathrm{T}})$$

调整矩阵顺序，有：

$$\boldsymbol{W}_d^* = \underset{\boldsymbol{W}_d}{\operatorname{argmin}} \operatorname{tr}(\boldsymbol{W}_d^{\mathrm{T}} \boldsymbol{X}\boldsymbol{X}^{\mathrm{T}} \boldsymbol{W}_d)$$

该最优化问题的求解就是求解 $\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}$ 的特征值。因此只需要对矩阵 $\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}$（也称为样本的协方差矩阵，它是一个 n 阶方阵）进行特征值分解，将求得的特征值排序：$\lambda_1 \geqslant \lambda_2 \geqslant \cdots \geqslant \lambda_n$，然后取前 d 个特征值对应的特征向量构成 $\boldsymbol{W} = (\boldsymbol{w}_1, \boldsymbol{w}_2, \cdots, \boldsymbol{w}_d)$。

3.2.2 PCA 算法

下面介绍 PCA 算法。

（1）输入：样本集 $D = \{\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_N\}$；低维空间维数 d。

（2）输出：投影矩阵 $\boldsymbol{W} = (\vec{w}_1, \vec{w}_2, \cdots, \vec{w}_d)$。

（3）算法步骤表现在：

- 对所有样本进行中心化操作

$$\vec{x}_i \leftarrow \vec{x}_i - \frac{1}{N} \sum_{j=1}^{N} \vec{x}_j$$

- 计算样本的协方差矩阵 $\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}$；
- 对协方差矩阵 $\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}$ 做特征值分解；
- 取最大的 d 个特征值对应的特征向量 $\vec{w}_1, \vec{w}_2, \cdots, \vec{w}_d$，构造投影矩阵 $\boldsymbol{W} = (\vec{w}_1, \vec{w}_2, \cdots, \vec{w}_d)$。

通常低维空间维数 d 的选取有两种方法：

- 通过交叉验证法选取较好的 d（在降维后的学习器的性能比较好）。
- 从算法原理的角度设置一个阈值，例如 $t = 95\%$，然后选取使得下式成立的最小的 d 的值：

$$\frac{\sum_{i=1}^{d} \lambda_i}{\sum_{i=1}^{n} \lambda_i} \geqslant t$$

其中，λ_i 从大到小排列。

3.2.3 PCA 降维的两个准则

PCA 降维的准则有以下两个：

- 最近重构性——就是前面介绍的样本集中所有点，重构后的点距离原来的点的误差之和最小。
- 最大可分性——样本点在低维空间的投影尽可能分开。

可以证明，最近重构性就等价于最大可分性。证明如下：对于样本点 $\vec{x}_i$，它在降维后空间中的投影是 $\vec{z}_i$。根据：

$$\hat{\vec{x}}_i = (\vec{w}_1, \vec{w}_2, \cdots, \vec{w}_d) \begin{bmatrix} z_i^{(1)} \\ z_i^{(2)} \\ \vdots \\ z_i^{(d)} \end{bmatrix} = \boldsymbol{W}\vec{z}_i$$

由投影矩阵的性质，以及$\hat{\vec{x}}_i$与$\vec{x}_i$的关系，有$\vec{z}_i = \boldsymbol{W}^{\mathrm{T}}\vec{x}_i$。

由于样本数据进行了中心化：即$\sum_i \vec{x}_i = (0,0,\cdots,0)^{\mathrm{T}}$，因此投影后，样本点的方差为：

$$\sum_{i=1}^{N} \boldsymbol{W}^{\mathrm{T}}\vec{x}_i\vec{x}_i^{\mathrm{T}}\boldsymbol{W}$$

令$\boldsymbol{X} = (\vec{x}_1,\vec{x}_2,\cdots,\vec{x}_N)$为$n \times N$维矩阵，于是根据样本点的方差最大，优化目标可为：

$$\max_{\boldsymbol{W}} \mathrm{tr}(\boldsymbol{W}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W})$$

$$\text{s.t.}\quad \boldsymbol{W}^{\mathrm{T}}\boldsymbol{W} = \boldsymbol{I}$$

这就是前面最后重构性推导的结果。

【例 3-1】 通过 Python 的 sklearn 库来实现鸢尾花数据进行降维，数据本身是四维的降维后变成二维，可以在平面中画出样本点的分布。样本数据结构如图 3-1 所示。

萼片长度	萼片宽度	花瓣长度	花瓣宽度	类别
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5	3.6	1.4	0.2	Iris-setosa
5.4	3.9	1.7	0.4	Iris-setosa
4.6	3.4	1.4	0.3	Iris-setosa
5	3.4	1.5	0.2	Iris-setosa
4.4	2.9	1.4	0.2	Iris-setosa
4.9	3.1	1.5	0.1	Iris-setosa
5.4	3.7	1.5	0.2	Iris-setosa
4.8	3.4	1.6	0.2	Iris-setosa
4.8	3	1.4	0.1	Iris-setosa
4.3	3	1.1	0.1	Iris-setosa
5.8	4	1.2	0.2	Iris-setosa

图 3-1　鸢尾花数据

其中样本总数为 150，鸢尾花的类别有 3 种，分别标记为 0、1、2。

```
import matplotlib.pyplot as plt                    # 加载 matplotlib 用于数据的可视化
from sklearn.decomposition import PCA              # 加载 PCA 算法包
from sklearn.datasets import load_iris

data = load_iris()
y = data.target
x = data.data
pca = PCA(n_components = 2)                        # 加载 PCA 算法，设置降维后主成分数目为 2
reduced_x = pca.fit_transform(x)                   # 对样本进行降维

red_x,red_y = [],[]
blue_x,blue_y = [],[]
green_x,green_y = [],[]

for i in range(len(reduced_x)):
    if y[i] == 0:
        red_x.append(reduced_x[i][0])
        red_y.append(reduced_x[i][1])
```

```
    elif y[i] == 1:
        blue_x.append(reduced_x[i][0])
        blue_y.append(reduced_x[i][1])
    else:
        green_x.append(reduced_x[i][0])
        green_y.append(reduced_x[i][1])
#可视化
plt.scatter(red_x,red_y,c = 'r',marker = 'x')
plt.scatter(blue_x,blue_y,c = 'b',marker = 'D')
plt.scatter(green_x,green_y,c = 'g',marker = '.')
plt.show()
```

运行程序,得到如图 3-2 所示的效果。

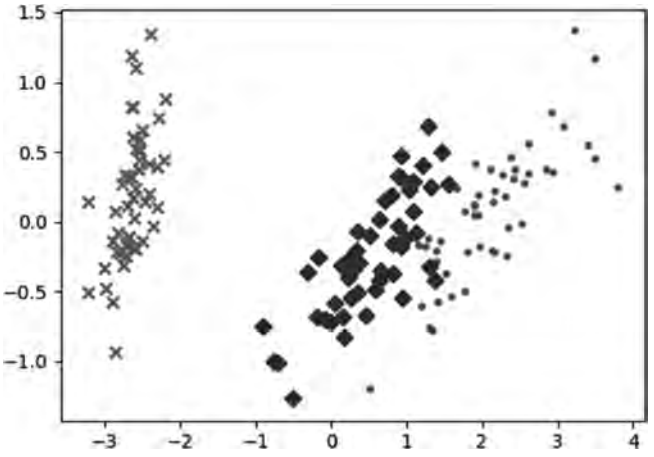

图 3-2 主成分降维效果

【例 3-2】 利用 PCA 对给定的数据 data2.txt 进行降维处理。

其实现步骤为:

(1) 首先引入 numpy,由于测试中用到了 pandas 和 matplotlib,所以这里一并加载。

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

(2) 定义一个均值函数。

```
#计算均值,要求输入数据为 numpy 的矩阵格式,行表示样本数,列表示特征
def meanX(dataX):
    return np.mean(dataX,axis = 0) #axis = 0 表示按照列来求均值,如果输入 list,则 axis = 1
```

(3) 编写 pca 方法,具体解释参考注释。

```
"""
```

参数:

- XMat——传入的是一个 numpy 的矩阵格式,行表示样本数,列表示特征。
- k——表示取前 k 个特征值对应的特征向量。

返回值：

- finalData——指的是返回的低维矩阵，对应于输入 reconData。
- reconData——对应的是移动坐标轴后的矩阵。

```
"""
def pca(XMat, k):
    average = meanX(XMat)
    m, n = np.shape(XMat)
    data_adjust = []
    avgs = np.tile(average, (m, 1))
    data_adjust = XMat - avgs
    covX = np.cov(data_adjust.T)                    #计算协方差矩阵
    featValue, featVec = np.linalg.eig(covX)        #求解协方差矩阵的特征值和特征向量
    index = np.argsort(-featValue)                  #按照 featValue 进行从大到小排序
    finalData = []
    if k > n:
        print("k must lower than feature number")
        return
    else:
  #注意特征向量时列向量，而 numpy 的二维矩阵(数组)a[m][n]中，a[1]表示第 1 行值
        selectVec = np.matrix(featVec.T[index[:k]])  #所以这里需要进行转置
        finalData = data_adjust * selectVec.T
        reconData = (finalData * selectVec) + average
    return finalData, reconData
```

(4) 编写一个加载数据集的函数。

```
#输入文件的每行数据都以\t 隔开
def loaddata(datafile):
    return np.array(pd.read_csv(datafile,sep = "\t",header = -1)).astype(np.float)
```

(5) 可视化结果。将维数 k 指定为 2，所以可以使用下面的函数将其绘制出来：

```
def plotBestFit(data1, data2):
    dataArr1 = np.array(data1)
    dataArr2 = np.array(data2)

    m = np.shape(dataArr1)[0]
    axis_x1 = []
    axis_y1 = []
    axis_x2 = []
    axis_y2 = []
    for i in range(m):
        axis_x1.append(dataArr1[i,0])
        axis_y1.append(dataArr1[i,1])
        axis_x2.append(dataArr2[i,0])
        axis_y2.append(dataArr2[i,1])
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.scatter(axis_x1, axis_y1, s = 50, c = 'red', marker = 's')
    ax.scatter(axis_x2, axis_y2, s = 50, c = 'blue')
```

```
    plt.xlabel('x1'); plt.ylabel('x2');
    plt.savefig("outfile.png")
    plt.show()
```

(6) 测试方法。将测试方法写入 main 函数中,然后直接执行 main 函数即可:

```
#根据数据集 data.txt
def main():
    datafile = "data2.txt"
    XMat = loaddata(datafile)
    k = 2
    return pca(XMat, k)
if __name__ == "__main__":
    finalData, reconMat = main()
    plotBestFit(finalData, reconMat)
```

运行程序,效果如图 3-3 所示。

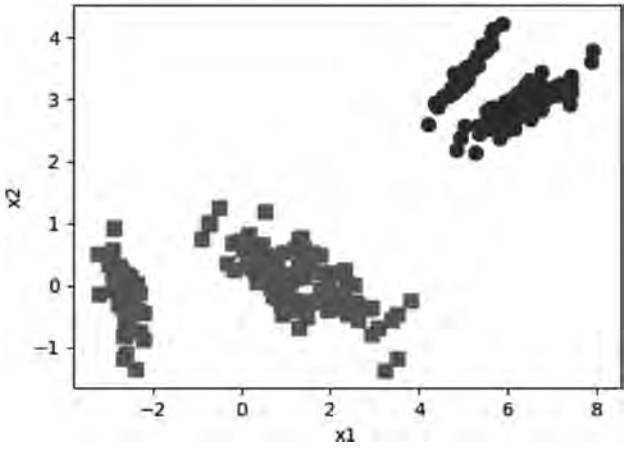

图 3-3 PCA 降维处理效果

3.3 SVD 降维

奇异值分解(SVD):设 $\boldsymbol{X}$ 为 $n\times N$ 阶矩阵,且 $\mathrm{rank}(\boldsymbol{X})=r$,则 n 阶正交矩阵 $\boldsymbol{V}$ 和 N 阶正交矩阵 $\boldsymbol{U}$,使得:

$$\boldsymbol{V}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{U}=\begin{bmatrix}\boldsymbol{\Sigma} & 0\\ 0 & 0\end{bmatrix}_{n\times N}$$

其中,

$$\boldsymbol{\Sigma}=\begin{bmatrix}\sigma_1 & 0 & \cdots & 0\\ 0 & \sigma_2 & \cdots & 0\\ \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & \cdots & \sigma_r\end{bmatrix}$$

其中,$\sigma_1\geqslant\sigma_2\geqslant\cdots\geqslant\sigma_r>0$。

根据正交矩阵的性质，$\boldsymbol{V}\boldsymbol{V}^{\mathrm{T}}=\boldsymbol{I}$，$\boldsymbol{U}\boldsymbol{U}^{\mathrm{T}}=\boldsymbol{I}$，有：

$$\boldsymbol{X}=\boldsymbol{V}\begin{bmatrix}\boldsymbol{\Sigma} & 0\\ 0 & 0\end{bmatrix}_{n\times N}\boldsymbol{U}^{\mathrm{T}}\Rightarrow\boldsymbol{X}^{\mathrm{T}}=\boldsymbol{U}\begin{bmatrix}\boldsymbol{\Sigma} & 0\\ 0 & 0\end{bmatrix}_{n\times N}\boldsymbol{V}^{\mathrm{T}}$$

则有 $\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}=\boldsymbol{V}\boldsymbol{M}\boldsymbol{V}^{\mathrm{T}}$，其中 $\boldsymbol{M}$ 是 n 阶对角矩阵：

$$\boldsymbol{M}=\begin{bmatrix}\boldsymbol{\Sigma} & 0\\ 0 & 0\end{bmatrix}_{n\times N}\begin{bmatrix}\boldsymbol{\Sigma} & 0\\ 0 & 0\end{bmatrix}_{N\times n}=\begin{bmatrix}\lambda_1 & 0 & 0 & \cdots & 0\\ 0 & \lambda_2 & 0 & \cdots & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & 0 & \cdots & \lambda_n\end{bmatrix}_{n\times n}$$

$$\lambda_i=\sigma_i^2 \quad i=1,2,\cdots,r$$

$$\lambda_i=0 \quad i=r+1,r+2,\cdots,n$$

于是有：$\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}=\boldsymbol{V}\boldsymbol{M}$。根据 $\boldsymbol{M}$ 是对角矩阵的性质，有 $\boldsymbol{V}\boldsymbol{M}=\boldsymbol{M}\boldsymbol{V}$，则有：

$$\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}=\boldsymbol{M}\boldsymbol{V}$$

则 $\lambda_i(i=1,2,\cdots,r)$就是 $\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}$ 的特征值，其对应的特征向量组成正交矩阵 $\boldsymbol{V}$。因此 SVD 奇异值分解等价于 PCA 主成分分析，核心都是求解 $\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}$ 的特征值以及对应的特征向量。

【例 3-3】 利用 SVD 对给定的数据进行降维处理。

```
import numpy as np
class CSVD(object):
    '''
    实现 SVD 分解降维应用示例的 Python 代码
    '''
    def __init__(self, data):
        self.data = data                    #用户数据
        self.S = []                         #用户数据矩阵的奇异值序列
        self.U = []                         #svd 后的单位正交向量
        self.VT = []                        #svd 后的单位正交向量
        self.k = 0                          #满足 self.p 的最小 k 值(k 表示奇异值的个数)
        self.SD = []                        #对角矩阵,对角线上元素是奇异值
    def _svd(self):
        '''
        用户数据矩阵的 SVD 奇异值分解
        '''
        self.U, self.S, self.VT = np.linalg.svd(self.data)
        return self.U, self.S, self.VT
    def _calc_k(self, percentge):
    '''确定 k 值:前 k 个奇异值的平方和占比 >= percentage, 求满足此条件的最小 k 值
        :param percentage, 奇异值平方和的占比的阈值
        :return 满足阈值 percentage 的最小 k 值
        '''
        self.k = 0
        #用户数据矩阵的奇异值序列的平方和
        total = sum(np.square(self.S))
        svss = 0                            #奇异值平方和
        for i in range(np.shape(self.S)[0]):
            svss += np.square(self.S[i])
            if (svss/total) >= percentge:
```

```
                self.k = i+1
                break
        return self.k

    def _buildSD(self, k):
        '''构建由奇异值组成的对角矩阵
        :param k,根据奇异值开放和的占比阈值计算出来的 k 值
        :return 由 k 个前奇异值组成的对角矩阵
        '''
        #方法 1:用数组乘方法
        self.SD = np.eye(self.k) * self.S[:self.k]
        #方法 2:用自定义方法
        e = np.eye(self.k)
        for i in range(self.k):
            e[i,i] = self.S[i]
        return self.SD

    def DimReduce(self, percentage):
        '''
        SVD 降维
        :param percentage, 奇异值开方和的占比阈值
        :return 降维后的用户数据矩阵
        '''
        #Step1:svd 奇异值分解
        self._svd()
        #Step2:计算 k 值
        self._calc_k(percentage)
        print('\n按照奇异值开方和占比阈值 percentage = %d, 求得降维的 k = %d' %(percentage,
self.k))
        #Step3:构建由奇异值组成的对角矩阵
        self._buildSD(self.k)
        k,U,SD,VT = self.k,self.U, self.SD, self.VT
     #Step4:按照 svd 分解公式对用户数据矩阵进行降维,得到降维压缩后的数据矩阵
        a = U[:len(U), :k]
        b = np.dot(SD, VT[:k, :len(VT)])
        newData = np.dot(a,b)
        return newData

def CSVD_manual():
    #训练数据集,用户对商品的评分矩阵,行为多个用户对单个商品的评分,列为用户对每个
    #商品的评分
    data = np.array([[5, 5, 0, 5],
                     [5, 0, 3, 4],
                     [3, 4, 0, 3],
                     [0, 0, 5, 3],
                     [5, 4, 4, 5],
                     [5, 4, 5, 5]])
    percentage = 0.9
    svdor = CSVD(data)
    ret = svdor.DimReduce(percentage)
    print('=====================================================')
```

```
        print('原始用户数据矩阵:\n', data)
        print('降维后的数据矩阵:\n', ret)
        print(' ========================================================= ')
if __name__ == '__main__':
    CSVD_manual()
```

运行程序,输出如下:

```
按照奇异值开方和占比阈值 percentage = 0, 求得降维的 k = 2
 ========================================================
原始用户数据矩阵:
 [[5 5 0 5]
 [5 0 3 4]
 [3 4 0 3]
 [0 0 5 3]
 [5 4 4 5]
 [5 4 5 5]]
降维后的数据矩阵:
 [[ 5.28849359  5.16272812  0.21491237  4.45908018]
 [ 3.27680994  1.90208543  3.74001972  3.80580978]
 [ 3.53241827  3.54790444 -0.13316888  2.89840405]
 [ 1.14752376 -0.64171368  4.94723586  2.3845504 ]
 [ 5.07268706  3.66399535  3.78868965  5.31300375]
 [ 5.10856595  3.40187905  4.6166049   5.58222363]]
 ========================================================
```

3.4 核主成分分析降维

PCA 方法假设从高维空间到低维空间的函数映射是线性的,但是在很多现实任务中,可能需要非线性映射才能找到合适的降维空间来降维。非线性降维的一种常用方法是基于核技巧对线性降维方法进行核化(kernelized)。核主成分分析(Kernelized PCA,KPCA)是对 PCA 的一种推广。

假定原始属性空间中的样本点 $\vec{x}_i$ 通过将 ϕ 映射到高维特征空间的坐标为 $\vec{x}_{i,\phi}$,即 $\vec{x}_{i,\phi}=\phi(\vec{x}_i)$。且假设高维特征空间是 n 维的,即:$\vec{x}_{i,\phi}\in\mathbf{R}^n$。

假定要将高维特征空间中的数据投影到低维空间中,投影矩阵 $\boldsymbol{W}$ 为 $n\times d$ 维矩阵,根据 PCA 推导的结果,要求解方程:

$$\boldsymbol{X}_\phi\boldsymbol{X}_\phi^{\mathrm{T}}\boldsymbol{W}=\lambda\boldsymbol{W}$$

其中,$\boldsymbol{X}_\phi=(\vec{x}_{1,\phi},\vec{x}_{2,\phi},\cdots,\vec{x}_{N,\phi})$ 为 $n\times N$ 维矩阵,于是有:

$$\left(\sum_{i=1}^{N}\phi(\vec{x}_i)\phi(\vec{x}_i)^{\mathrm{T}}\right)\boldsymbol{W}=\lambda\boldsymbol{W}$$

通常并不清楚 ϕ 的解析表达式,于是引入核函数:

$$k(\vec{x}_i,\vec{x}_j)=\phi(\vec{x}_i)^{\mathrm{T}}\phi(\vec{x}_j)$$

定义核矩阵:

$$\boldsymbol{K}=\begin{bmatrix} k(\vec{x}_1,\vec{x}_1) & k(\vec{x}_1,\vec{x}_2) & \cdots & k(\vec{x}_1,\vec{x}_N) \\ k(\vec{x}_2,\vec{x}_1) & k(\vec{x}_2,\vec{x}_2) & \cdots & k(\vec{x}_2,\vec{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ k(\vec{x}_N,\vec{x}_1) & k(\vec{x}_N,\vec{x}_2) & \cdots & k(\vec{x}_N,\vec{x}_N) \end{bmatrix}$$

则有 $\boldsymbol{X}_\phi^{\mathrm{T}}\boldsymbol{X}_\phi=\boldsymbol{K}$。

定义：

$$\vec{\alpha}_i=\frac{\vec{x}_{i,\phi}^{\mathrm{T}}\boldsymbol{W}}{\lambda}$$

则 $\vec{\alpha}_i$ 为 $1\times d$ 维行向量。

定义：$\boldsymbol{A}=(\vec{\alpha}_1,\vec{\alpha}_2,\cdots,\vec{\alpha}_N)^{\mathrm{T}}$ 为 $N\times d$ 维矩阵，则有：

$$\boldsymbol{W}=\frac{1}{\lambda}\left(\sum_{i=1}^{N}\vec{x}_{i,\phi}\vec{x}_{i,\phi}^{\mathrm{T}}\right)\boldsymbol{W}=\sum_{i=1}^{N}\vec{x}_{i,\phi}\frac{\vec{x}_{i,\phi}^{\mathrm{T}}\boldsymbol{W}}{\lambda}=\sum_{i=1}^{N}\vec{x}_{i,\phi}\vec{\alpha}_i=\boldsymbol{X}_\phi\boldsymbol{A}$$

将 $\boldsymbol{W}=\boldsymbol{X}_\phi\boldsymbol{A}$ 代入

$$\boldsymbol{X}_\phi\boldsymbol{X}_\phi^{\mathrm{T}}\boldsymbol{W}=\lambda\boldsymbol{W}$$

得到：

$$\boldsymbol{X}_\phi\boldsymbol{X}_\phi^{\mathrm{T}}\boldsymbol{X}_\phi\boldsymbol{A}=\lambda\boldsymbol{X}_\phi\boldsymbol{A}$$

两边同时左乘以 $\boldsymbol{X}_\phi^{\mathrm{T}}$，再代入 $\boldsymbol{X}_\phi^{\mathrm{T}}\boldsymbol{X}_\phi=\boldsymbol{K}$，有：

$$\boldsymbol{K}\boldsymbol{K}\boldsymbol{A}=\lambda\boldsymbol{K}\boldsymbol{A}$$

如果要求核矩阵可逆，则上式两边同时左乘以 $\boldsymbol{K}^{-1}$，则有：

$$\boldsymbol{K}\boldsymbol{A}=\lambda\boldsymbol{A}$$

同样该问题也是一个特征值分解问题，取 $\boldsymbol{K}$ 最大的 d 个特征值对应的特征向量组成 $\boldsymbol{W}$ 即可。

对于新样本 $\vec{x}$，其投影后第 $j(j=1,2,\cdots,d)$ 维的坐标为：

$$z_j=\boldsymbol{w}_j^{\mathrm{T}}\phi(\vec{x})=\sum_{i=1}^{N}\alpha_i^{(j)}\phi(\vec{x}_i)^{\mathrm{T}}\phi(\vec{x})=\sum_{i=1}^{N}\alpha_i^{(j)}k(\vec{x}_i,\vec{x})$$

其中，$\alpha_i^{(j)}$ 为行向量 $\vec{\alpha}_i$ 的第 j 个分量。可以看到，为了获取投影后的坐标，KPCA 需要对所有样本求和，因此它的计算开销更大。

【例 3-4】 对数据实现非线性映射降维(KPCA 方法)。

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import pdist,squareform
from scipy import exp
from scipy.linalg import eigh
from sklearn.datasets import make_moons
from sklearn.datasets import make_circles
from sklearn.decomposition import PCA
from matplotlib.ticker import FormatStrFormatter
def rbf_kernel_pca(X,gama,n_components):
    #1:计算样本对欧几里得距离,并生成核矩阵 k(x,y) = exp( - gama * ||x - y||^2),x 和 y 表示
```

```
    #样本,构建一个NXN的核矩阵,矩阵值是样本间的欧氏距离值。计算两两样本间欧几里得距离
    sq_dists = pdist(X, 'sqeuclidean')
    ##距离平方
    mat_sq_dists = squareform(sq_dists)
    ##计算对称核矩阵
    K = exp(-gama * mat_sq_dists)
    #2:聚集核矩阵K' = K-L*K-K*L + L*K*L,其中L是一个n×n的矩阵(和核矩阵K的维数
    #同,所有的值都是1/n。聚集核矩阵的必要性是:样本经过标准化处理后,当在生成协方差
    #并以非线性特征的组合替代点积时,所有特征的均值为0;但用低维点积计算时并没有精
    #算新的高维特征空间,也无法确定新特征空间的中心在零点
    N = K.shape[0]
    one_n = np.ones((N,N))/N                    #N×N单位矩阵
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)
    #3:对聚集后的核矩阵求取特征值和特征向量
    eigvals,eigvecs = eigh(K)

    #4:选择前K个特征值所对应的特征向量,和PCA不同,KPCA得到的K个特征,不是主成分轴,而
    #是高维映射到低维后的低维特征数量核化过程是低维映射到高维,PCA是降维,经过核化后的
    #维度已经不是原来的特征空间。核化是低维映射到高维,但并不是在高维空间计算(非线性特
    #征组合)而是在低维空间计算(点积),做到这点关键是核函数,核函数通过两个向量点积来度
    #量向量间相似度,能在低维空间内近似计算出高维空间的非线性特征空间
    X_pc = np.column_stack((eigvecs[:,-i] for i in range(1,n_components+1)))
    return X_pc
###分离半月形数据
##生成二维线性不可分数据
X,y = make_moons(n_samples = 100,random_state = 123)
plt.scatter(X[y == 0,0],X[y == 0,1],color = 'red',marker = '^',alpha = 0.5)
plt.scatter(X[y == 1,0],X[y == 1,1],color = 'blue',marker = 'o',alpha = 0.5)
plt.show()
##PCA降维,映射到主成分,仍不能很好地进行线性分类
sk_pca = PCA(n_components = 2)
X_spca = sk_pca.fit_transform(X)
fig,ax = plt.subplots(nrows = 1,ncols = 2,figsize = (7,3))
ax[0].scatter(X_spca[y == 0,0],X_spca[y == 0,1],color = 'red',marker = '^',alpha = 0.5)
ax[0].scatter(X_spca[y == 1,0],X_spca[y == 1,1],color = 'blue',marker = 'o',alpha = 0.5)
ax[1].scatter(X_spca[y == 0,0],np.zeros((50,1)) + 0.02,color = 'red',marker = '^',alpha = 0.5)
ax[1].scatter(X_spca[y == 1,0],np.zeros((50,1)) - 0.02,color = 'blue',marker = '^',alpha = 0.5)
ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([-1,1])
ax[1].set_yticks([])
ax[1].set_xlabel('PC1')
plt.show()
##利用基于RBF核的KPCA来实现线性可分
X_kpca = rbf_kernel_pca(X, gama = 15, n_components = 2)
fig,ax = plt.subplots(nrows = 1,ncols = 2,figsize = (7,3))
ax[0].scatter(X_kpca[y == 0,0],X_kpca[y == 0,1],color = 'red',marker = '^',alpha = 0.5)
ax[0].scatter(X_kpca[y == 1,0],X_kpca[y == 1,1],color = 'blue',marker = 'o',alpha = 0.5)
```

```
ax[1].scatter(X_kpca[y == 0,0],np.zeros((50,1)) + 0.02,color = 'red',marker = '^',alpha = 0.5)
ax[1].scatter(X_kpca[y == 1,0],np.zeros((50,1)) - 0.02,color = 'blue',marker = '^',alpha = 0.5)
ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([ - 1,1])
ax[1].set_yticks([])
ax[1].set_xlabel('PC1')
ax[0].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
ax[1].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
plt.show()

###分离同心圆
##生成同心圆数据
X,y = make_circles(n_samples = 1000,random_state = 123,noise = 0.1,factor = 0.2)
plt.scatter(X[y == 0,0],X[y == 0,1],color = 'red',marker = '^',alpha = 0.5)
plt.scatter(X[y == 1,0],X[y == 1,1],color = 'blue',marker = 'o',alpha = 0.5)
plt.show()
##标准 PCA 映射
sk_pca = PCA(n_components = 2)
X_spca = sk_pca.fit_transform(X)
fig,ax = plt.subplots(nrows = 1,ncols = 2,figsize = (7,3))
ax[0].scatter(X_spca[y == 0,0],X_spca[y == 0,1],color = 'red',marker = '^',alpha = 0.5)
ax[0].scatter(X_spca[y == 1,0],X_spca[y == 1,1],color = 'blue',marker = 'o',alpha = 0.5)
ax[1].scatter(X_spca[y == 0,0],np.zeros((500,1)) + 0.02,color = 'red',marker = '^',alpha = 0.5)
ax[1].scatter(X_spca[y == 1,0],np.zeros((500,1)) - 0.02,color = 'blue',marker = '^',alpha = 0.5)
ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([ - 1,1])
ax[1].set_yticks([])
ax[1].set_xlabel('PC1')
plt.show()
##RBF - KPCA 映射
X_kpca = rbf_kernel_pca(X, gama = 15, n_components = 2)
fig,ax = plt.subplots(nrows = 1,ncols = 2,figsize = (7,3))
ax[0].scatter(X_kpca[y == 0,0],X_kpca[y == 0,1],color = 'red',marker = '^',alpha = 0.5)
ax[0].scatter(X_kpca[y == 1,0],X_kpca[y == 1,1],color = 'blue',marker = 'o',alpha = 0.5)
ax[1].scatter(X_kpca[y == 0,0],np.zeros((500,1)) + 0.02,color = 'red',marker = '^',alpha = 0.5)
ax[1].scatter(X_kpca[y == 1,0],np.zeros((500,1)) - 0.02,color = 'blue',marker = '^',alpha = 0.5)
ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([ - 1,1])
ax[1].set_yticks([])
ax[1].set_xlabel('PC1')
ax[0].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
ax[1].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
plt.show()
```

运行程序,效果如图 3-4 所示。

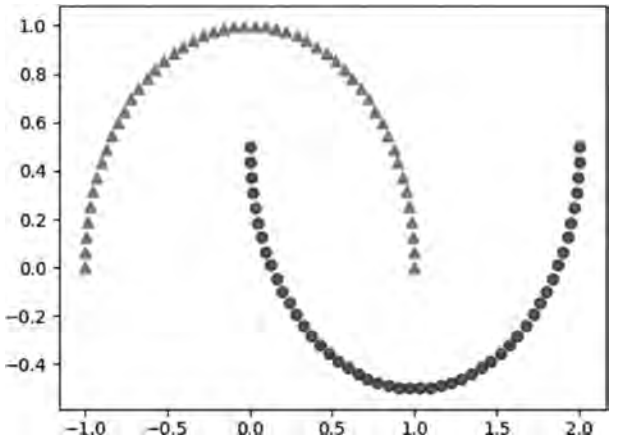

图 3-4 KPCA 降维效果

3.5 流形学习降维

流形学习(Manifold Learning)是一类借鉴了拓扑流形概念的降维方法,被认为属于非线性降维的一个分支。流形学习假设所处理的数据点分布在嵌入于欧氏空间的一个潜在的流形体上,或者说这些数据点可以构成这样一个潜在的流形体。流形学习的方法有很多,并具有一些共同的特征:首先需要构造流形上样本点的局部邻域结构,然后用这些局部邻域结构将样本点全局地映射到一个降维空间。这些方法之间的不同之处主要在于构造的局部邻域结构不同,以及利用这些局部邻域结构来构造全局的低维嵌入方法的不同。

3.6 多维缩放降维

3.6.1 原理

多维缩放(Multiple Dimensional Scaling,MDS)要求原始空间中样本之间的距离在低维空间中得到保持。

假设 N 个样本在原始空间中的距离矩阵为 $\boldsymbol{D}=(d_{i,j})_{N\times N}$:

$$\boldsymbol{D}=\begin{bmatrix} d_{1,1} & d_{1,2} & \cdots & d_{1,N} \\ d_{2,1} & d_{2,2} & \cdots & d_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ d_{N,1} & d_{N,2} & \cdots & d_{N,N} \end{bmatrix}$$

其中,$d_{i,j}=\|\vec{x}_i-\vec{x}_j\|$ 为样本 $\vec{x}_i$ 到样本 $\vec{x}_j$ 的距离。

假设原始样本是在 n 维空间,我们的目标是在 n,$n'<n$ 维空间里获取样本,欧氏距离保持不变。

假设样本集在原空间的表示 $\boldsymbol{X}=(\vec{x}_1,\vec{x}_2,\cdots,\vec{x}_N)$ 为 $n\times N$ 维矩阵,样本集在降维后空间的坐标 $\boldsymbol{Z}=(\vec{z}_1,\vec{z}_2,\cdots,\vec{z}_N)$ 为 $n'\times N$ 维矩阵。所求的正是 $\boldsymbol{Z}$ 矩阵。

令 $\boldsymbol{B}=\boldsymbol{Z}^{\mathrm{T}}\boldsymbol{Z}$ 为 $N\times N$ 维矩阵，即

$$\boldsymbol{B}=\begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,N} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ b_{N,1} & b_{N,2} & \cdots & b_{N,N} \end{bmatrix}$$

其中，$b_{i,j}=\vec{z}_i\cdot\vec{z}_j$ 为降维后样本的内积。

根据降维前后样本的欧氏距离保持不变，有：

$$d_{i,j}^2=\|\vec{z}_i-\vec{z}_j\|^2=\|\vec{z}_i\|^2+\|\vec{z}_j\|^2-2\vec{z}_i^{\mathrm{T}}\vec{z}_j=b_{i,i}+b_{j,j}-2b_{i,j}$$

假设降维后的样本集 $\boldsymbol{Z}$ 被中心化，即 $\sum_{i=1}^{N}\vec{z}_i=\vec{0}$，则矩阵 $\boldsymbol{B}$ 的每行之和均为零，每列之和均为零，即：

$$\sum_{i=1}^{N}b_{i,j}=0,\quad j=1,2,\cdots,N$$

$$\sum_{j=1}^{N}b_{i,j}=0,\quad i=1,2,\cdots,N$$

于是有：

$$\sum_{i=1}^{N}d_{i,j}^2=\sum_{i=1}^{N}b_{i,i}+Nb_{j,j}=\mathrm{tr}(\boldsymbol{B})+Nb_{j,j}$$

$$\sum_{j=1}^{N}d_{i,j}^2=\sum_{i=1}^{N}b_{j,j}+Nb_{i,i}=\mathrm{tr}(\boldsymbol{B})+Nb_{i,i}$$

$$\sum_{i=1}^{N}\sum_{i=1}^{N}d_{i,j}^2=\sum_{i=1}^{N}(\mathrm{tr}(\boldsymbol{B})+Nb_{i,i})=2N\,\mathrm{tr}(\boldsymbol{B})$$

其中，$\mathrm{tr}(B)$表示矩阵 $\boldsymbol{B}$ 的迹。

令

$$d_{i,\cdot}^2=\frac{1}{N}\sum_{i=1}^{N}d_{ij}^2=\frac{\mathrm{tr}(\boldsymbol{B})}{N}+b_{i,i}$$

$$d_{j,\cdot}^2=\frac{1}{N}\sum_{j=1}^{N}d_{ij}^2=\frac{\mathrm{tr}(\boldsymbol{B})}{N}+b_{j,j}$$

$$d_{\cdot,\cdot}^2=\frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N}d_{ij}^2=\frac{2\mathrm{tr}(\boldsymbol{B})}{N}$$

代入 $d_{i,j}^2=b_{i,i}+b_{j,j}-2b_{i,j}$，有

$$b_{i,j}=\frac{b_{i,i}+b_{j,j}-d_{i,j}^2}{2}=\frac{d_{i,\cdot}^2+d_{j,\cdot}^2-d_{\cdot,\cdot}^2-d_{i,j}^2}{2}$$

上面右式根据 $d_{i,j}$ 给出了 $b_{i,j}$，因此可以根据原始空间中的距离矩阵 $\boldsymbol{D}$ 求出在降维后空间的内积矩阵 $\boldsymbol{B}$。现在的问题是：已知内积矩阵 $\boldsymbol{B}=\boldsymbol{Z}^{\mathrm{T}}\boldsymbol{Z}$，如何求得矩阵 $\boldsymbol{Z}$。

对矩阵 $\boldsymbol{B}$ 做特征值分解，设 $\boldsymbol{B}=\boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^{\mathrm{T}}$，其中$\boldsymbol{\Lambda}=\mathrm{diag}(\lambda_1,\lambda_2,\cdots,\lambda_N)$为特征值构成的对角矩阵，其中 $\lambda_1\geqslant\lambda_2\geqslant\cdots\geqslant\lambda_N$，$\boldsymbol{V}$ 为特征向量矩阵。

假定特征中有 n^* 个非零特征值，它们构成对角矩阵$\boldsymbol{\Lambda}_*=\mathrm{diag}(\lambda_1,\lambda_2,\cdots,\lambda_{n^*})$。令 $\boldsymbol{V}_*$ 为对应的特征向量矩阵，则

$$\boldsymbol{Z}=\boldsymbol{\Lambda}_{*}^{1/2}\boldsymbol{V}_{*}^{\mathrm{T}}$$

其中，$\boldsymbol{Z}$ 为 $n^{*}\times N$ 阶矩阵，此时有 $n'=n^{*}$。

在现实应用中，为了有效降维，往往仅需要降维后的距离与原始空间中的距离尽可能相等，而不必严格相等。此时可以取 $n'\ll n^{*}\leqslant n$ 个最大特征值构成的对角矩阵为：

$$\widetilde{\boldsymbol{\Lambda}}=\mathrm{diag}(\lambda_1,\lambda_2,\cdots,\lambda_{n'})$$

令 $\widetilde{\boldsymbol{V}}$ 表示对应的特征向量矩阵，则

$$\boldsymbol{Z}=\widetilde{\boldsymbol{\Lambda}}^{1/2}\boldsymbol{V}^{\mathrm{T}}\in\mathbf{R}^{n'\times N}$$

3.5.2 MDS 算法

多维缩放(MDS)算法如下。

(1) 输入：距离矩阵 $\boldsymbol{D}\in\mathbf{R}^{N\times N}$；低维空间维数 n'。

(2) 输出：样本集在低维空间中的矩阵 $\boldsymbol{Z}$。

(3) 算法步骤为：

- 根据下列式子计算 $d_{i,\cdot}^2$、$d_{j,\cdot}^2$、$d_{\cdot,\cdot}^2$。

$$d_{i,\cdot}^2=\frac{1}{N}\sum_{i=1}^{N}d_{i,j}^2=\frac{\mathrm{tr}(\boldsymbol{B})}{N}+b_{i,i}$$

$$d_{j,\cdot}^2=\frac{1}{N}\sum_{j=1}^{N}d_{i,j}^2=\frac{\mathrm{tr}(\boldsymbol{B})}{N}+b_{j,j}$$

$$d_{\cdot,\cdot}^2=\frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N}d_{i,j}^2=\frac{2\mathrm{tr}(\boldsymbol{B})}{N}$$

- 根据正式计算矩阵 $\boldsymbol{B}$：

$$b_{i,j}=\frac{b_{i,i}+b_{j,j}-d_{i,j}^2}{2}=\frac{d_{i,\cdot}^2+d_{j,\cdot}^2-d_{\cdot,\cdot}^2-d_{i,j}^2}{2}$$

- 对矩阵 $\boldsymbol{B}$ 进行特征值分解。

- $\widetilde{\boldsymbol{\Lambda}}$ 为 n' 个最大特征值所构成的对角矩阵，$\widetilde{\boldsymbol{V}}$ 表示对应的特征向量矩阵，则：

$$\boldsymbol{Z}=\widetilde{\boldsymbol{\Lambda}}^{1/2}\boldsymbol{V}^{\mathrm{T}}\in\mathbf{R}^{n'\times N}$$

【例 3-5】 利用 MDS 算法实现数据的降维。

```
"""
    MDS 降维
"""
import numpy as np
import matplotlib.pyplot as plt
from sklearn import    datasets,manifold
def load_data():
    '''
    加载用于降维的数据
    :return: 一个元组,依次为训练样本集和样本集的标记
    '''
    iris = datasets.load_iris()              # 使用 scikit-learn 自带的 iris 数据集
```

```
    return  iris.data,iris.target
def test_MDS(*data):
    '''
    测试 MDS 的用法
    :param data: 可变参数.它是一个元组,这里要求其元素依次为:训练样本集、训练样本的标记
    '''
    X,y = data
    for n in [4,3,2,1]:                          #依次考查降维目标为四维、三维、二维、一维
        mds = manifold.MDS(n_components = n)
        mds.fit(X)
        print('stress(n_components = %d) : %s'% (n, str(mds.stress_)))
def plot_MDS(*data):
    '''
    绘制经过 使用 MDS 降维到二维之后的样本点
    :param data: 可变参数.它是一个元组,这里要求其元素依次为:训练样本集、训练样本的标记
    '''
    X,y = data
    mds = manifold.MDS(n_components = 2)
    X_r = mds.fit_transform(X)                   #原始数据集转换到二维
    ###绘制二维图形
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    colors = ((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
        (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)   #颜色集合,不同标记的样本
                                                              #染不同的颜色
    for label ,color in zip( np.unique(y),colors):
        position = y == label
        ax.scatter(X_r[position,0],X_r[position,1],label = "target =  %d" % label,color =
color)
    ax.set_xlabel("X[0]")
    ax.set_ylabel("X[1]")
    ax.legend(loc = "best")
    ax.set_title("MDS")
    plt.show()
if __name__ == '__main__':
    X,y = load_data()                            #产生用于降维的数据集
    test_MDS(X,y)                                #调用 test_MDS
    plot_MDS(X,y)                                #调用 plot_MDS
```

运行程序,输出如下,效果如图 3-5 所示。

```
stress(n_components = 4) : 11.887221490372065
stress(n_components = 3) : 27.590871404910008
stress(n_components = 2) : 113.30127128702554
stress(n_components = 1) : 28321.42085807291
```

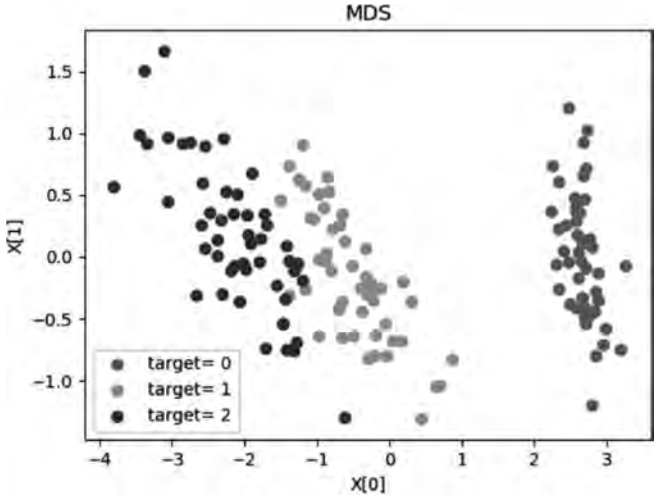

图 3-5 MDS 降维效果

3.7 等度量映射降维

等度量映射(Isometric Mapping,Isomap)原理如下：

(1) 首先建立近邻连接图：利用流形在局部上与欧氏空间同胚这个性质，基于欧氏距离，对每个点找出它在低维流形上的近邻点，建立近邻连接图。

(2) 计算任意两点之间的距离：计算近邻连接图上任意两点之间的最短路径问题，作为两点之间的距离。

(3) 在得到任意两点的距离之后，就可以通过 MDS 算法来获得样本点在低维空间中的坐标。

Isomap 算法如下。

(1) 输入：样本集 $\boldsymbol{D}=\{\vec{x}_1,\vec{x}_2,\cdots,\vec{x}_N\}$；近邻参数 k；低维空间维数 n'。

(2) 输出：样本集在低维空间中的矩阵 $\boldsymbol{Z}$。

(3) 算法步骤为：

- 对每个样本点 $\vec{x}_i$，计算它的 k 近邻；同时将 $\vec{x}_i$ 与它的 k 近邻的距离设置为欧氏距离，与其他点的距离设置为无穷大。
- 调用最短路径算法计算任意两个样本点之间的距离，获得距离矩阵 $\boldsymbol{D}\in\mathbf{R}^{N\times N}$。
- 调用多维缩放 MDS 算法，获得样本集在低维空间中的矩阵 $\boldsymbol{Z}$。

Isomap 算法有个很大的问题：对于新样本，难以将其映射到低维空间。理论上可以将新样本添加到样本集中，重新调用 Isomap 算法，这种方案计算量太大。一般的解决方法是：训练一个回归学习器来对新样本的低维空间进行预测。

近邻图有如下两种类型。

(1) k 近邻图：指定近邻点个数，如指定距离最近的 k 个点为近邻点。

(2) ε 近邻图：指定距离阈值 ε，距离小于 ε 的点被认为是近邻点。

在建立近邻图的时候要注意控制近邻图的范围，否则容易出现“短路”或者“断路”问题。

(1)“短路”问题：近邻范围指定过大，距离很远的点也被误认为是近邻。

(2)“断路”问题：近邻范围指定过小，本应该相连的区域被认为是断开的。

【例 3-6】 利用等度量映射(Isomap)算法对数据进行降维。

(1) 导入必要的编程库。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets,decomposition,manifold
```

(2) 加载数据。

```
def load_data():
    iris = datasets.load_iris()
    return iris.data,iris.target
```

(3) 使用 somap od。

```
def test_Isomap(*data):
    X,y = data
    for n in [4,3,2,1]:
        isomap = manifold.Isomap(n_components = n)
        isomap.fit(X)
        print('reconstruction_error(n_components = %d): %s' %(n,
            isomap.reconstruction_error()))
X,y = load_data()
test_Isomap(X,y)
```

(4) 降维后的样本分布图。

```
def plot_Isomap(*data):
    X,y = data
    Ks = [1,5,25,y.size-1]
    fig = plt.figure()
    for i,k in enumerate(Ks):
        isomap = manifold.Isomap(n_components = 2,n_neighbors = k)
        X_r = isomap.fit_transform(X)
        ax = fig.add_subplot(2,2,i+1)
        colors = ((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
                  (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)
        for label,color in zip(np.unique(y),colors):
            position = y==label

ax.scatter(X_r[position,0],X_r[position,1],label = 'target = %d' %label,color = color)
        ax.set_xlabel('X[0]')
        ax.set_ylabel('X[1]')
        ax.legend(loc = 'best')
        ax.set_title("k = %d" %k)
    plt.suptitle('Isomap')
    plt.show()
plot_Isomap(X,y)
```

（5）将原始数据的特征直接压缩到一维。

```
def plot_Isomap_k_d1( * data):
    X,y = data
    Ks = [1,5,25,y.size - 1]
    fig = plt.figure()
    for i,k in enumerate(Ks):
        isomap = manifold.Isomap(n_components = 2,n_neighbors = k)
        X_r = isomap.fit_transform(X)
        ax = fig.add_subplot(2,2,i + 1)
        colors = ((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
                  (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)
        for label,color in zip(np.unique(y),colors):
            position = y == label
ax.scatter(X_r[position],np.zeros_like(X_r[position]),label = 'target = % d' % label,color = color)
        ax.set_xlabel('X[0]')
        ax.set_ylabel('Y')
        ax.legend(loc = 'best')
        ax.set_title("k = % d" % k)
    plt.suptitle('Isomap')
    plt.show()
plot_Isomap_k_d1(X,y)
```

运行程序，输出如下，效果如图 3-6 和图 3-7 所示。

```
reconstruction_error(n_components = 4):1.0097180068081741
reconstruction_error(n_components = 3):1.0182845146289834
reconstruction_error(n_components = 2):1.0276983764330463
reconstruction_error(n_components = 1):1.0716642763207656
```

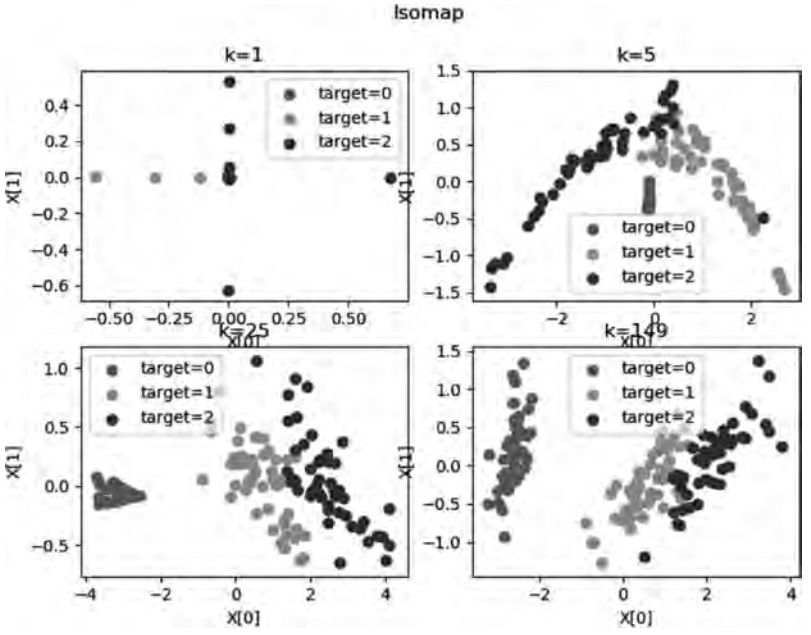

图 3-6　降维后的样本分布图

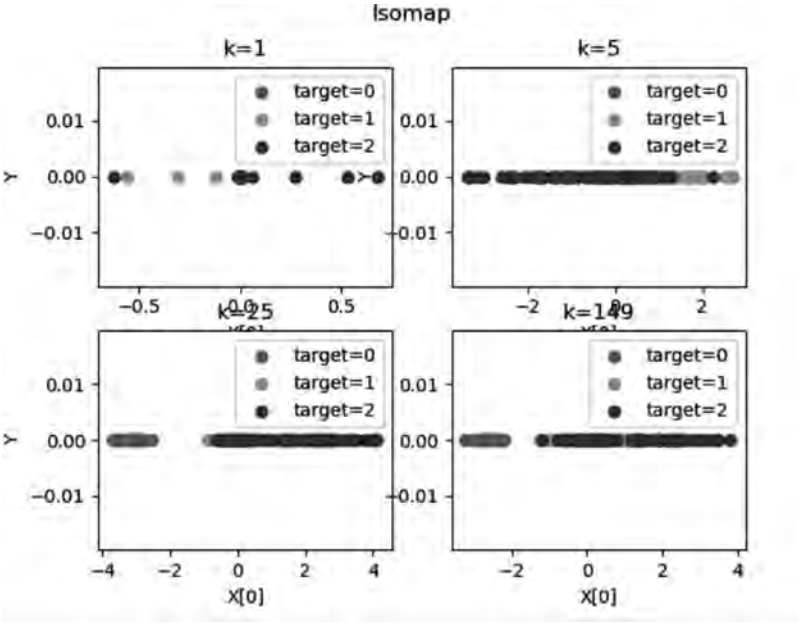

图 3-7 将原始数据的特征直接压缩到一维效果

3.8 局部线性嵌入

3.8.1 原理

局部线性嵌入(Locally Linear Embedding,LLE)的目标是：保持邻域内样本之间的线性关系。

对每个样本 $\vec{x}_i$，首先寻找其近邻点，假设这些近邻点的下标集合为 Q_i。然后需要计算基于 $\vec{x}_i$ 的近邻点对 $\vec{x}_j$ 进行线性重构的系数 $\vec{w}_i$。定义样本集重构误差为：

$$\text{err}=\sum_{i=1}^{N}\left\|\vec{x}_i-\sum_{j\in Q_i}w_{i,j}\vec{x}_j\right\|_2^2$$

其中，$w_{i,j}$ 为 $\vec{w}_i$ 的分量。我们的目标是使样本集重构误差最小，即：

$$\min_{\vec{w}_1,\vec{w}_2,\cdots,\vec{w}_N}\sum_{i=1}^{N}\left\|\vec{x}_i-\sum_{j\in Q_i}w_{i,j}\vec{x}_j\right\|_2^2$$

这样的解有无数个，对权重进行归一化处理，即：

$$\sum_{j\in Q_i}w_{i,j}=1,\quad i=1,2,\cdots,N$$

这样一来，就是求解最优化问题：

$$\min_{\vec{w}_1,\vec{w}_2,\cdots,\vec{w}_N}\sum_{i=1}^{N}\left\|\vec{x}_i-\sum_{j\in Q_i}w_{i,j}\vec{x}_j\right\|_2^2$$

$$\text{s.t.}\ \sum_{j\in Q_i}w_{i,j}=1,\quad i=1,2,\cdots,N$$

该最优化问题有解析解。令 $\boldsymbol{C}_{j,k}=(\vec{x}_i-\vec{x}_j)^{\mathrm{T}}(\vec{x}_i-\vec{x}_k)$，则可以解出：

$$w_{i,j}=\frac{\sum\limits_{k\in Q_i}\boldsymbol{C}_{j,k}^{-1}}{\sum\limits_{l,s\in Q_i}\boldsymbol{C}_{l,s}^{-1}},\quad j\in Q_i$$

求出了线性重构的系数 $\vec{w}_i$ 之后，LLE 在低维空间中保持 $\vec{w}_i$ 不变。设 $\vec{z}_i$ 对应的低维坐标为 $\vec{z}_j$，已知线性重构的系数 $\vec{w}_i$，定义样本集在低维空间中重构误差为：

$$\mathrm{err}'=\sum_{i=1}^{N}\left\|\vec{z}_i-\sum_{j\in Q_i}w_{i,j}\vec{z}_j\right\|_2^2$$

现在的问题是要求出 $\vec{z}_i$，从而使上式最小。即求解：

$$\min_{\vec{z}_1,\vec{z}_2,\cdots,\vec{z}_N}\sum_{i=1}^{N}\left\|\vec{z}_i-\sum_{j\in Q_i}w_{i,j}\vec{z}_j\right\|_2^2$$

令 $\boldsymbol{Z}=(\vec{z}_1,\vec{z}_2,\cdots,\vec{z}_N)\in\mathbf{R}^{n'\times N}$，其中 n' 为低维空间的维数（n 为原始样本所在的高维空间的维数）。令：

$$\boldsymbol{W}=\begin{bmatrix}w_{1,1}&w_{1,2}&\cdots&w_{1,N}\\w_{2,1}&w_{2,2}&\cdots&w_{2,N}\\\vdots&\vdots&\ddots&\vdots\\w_{N,1}&w_{N,2}&\cdots&w_{N,N}\end{bmatrix}$$

定义 $\boldsymbol{M}=(\boldsymbol{I}-\boldsymbol{W})^{\mathrm{T}}(\boldsymbol{I}-\boldsymbol{W})$，于是最优化问题可重写为：

$$\min_{\boldsymbol{Z}}\mathrm{tr}(\boldsymbol{Z}\boldsymbol{M}\boldsymbol{Z}^{\mathrm{T}})$$

该最优化问题有无数个解。添加约束 $\boldsymbol{Z}\boldsymbol{Z}^{\mathrm{T}}=\boldsymbol{I}$，于是最优化问题为：

$$\min_{\boldsymbol{Z}}\mathrm{tr}(\boldsymbol{Z}\boldsymbol{M}\boldsymbol{Z}^{\mathrm{T}})$$

$$\text{s.t.}\quad \boldsymbol{Z}\boldsymbol{Z}^{\mathrm{T}}=\boldsymbol{I}$$

该最优化问题可以通过特征值分解求解：选取 $\boldsymbol{M}$ 最小的 n' 个特征值对应的特征向量组成的矩阵即为 $\boldsymbol{Z}^{\mathrm{T}}$。

3.8.2 LLE 算法

LLE 算法如下。

（1）输入：样本集 $\boldsymbol{D}=\{\vec{x}_1,\vec{x}_2,\cdots,\vec{x}_N\}$；近邻参数 k；低维空间维数 n'。

（2）输出：样本集在低维空间中的矩阵 $\boldsymbol{Z}$。

（3）算法步骤：

- 对于样本集中的每个点 $\vec{x}_i$，$i=1,2,\cdots,N$，执行下列操作。
  - ◎ 确定 $\vec{x}_i$ 的 k 近邻，获得其近邻下标集合 Q_i。
  - ◎ 对于 $j\in Q_i$，根据下式计算 $w_{i,j}$。

$$w_{i,j}=\frac{\sum\limits_{k\in Q_i}C_{j,k}^{-1}}{\sum\limits_{l,s\in Q_i}C_{l,s}^{-1}}$$

$$\boldsymbol{C}_{j,k}=(\vec{x}_i-\vec{x}_j)^{\mathrm{T}}(\vec{x}_i-\vec{x}_k)$$

◎ 对于 $j\notin Q_i$，$w_{i,j}=0$。

- 根据 $w_{i,j}$ 构建矩阵 $\boldsymbol{W}$。
- 计算 $\boldsymbol{M}=(\boldsymbol{I}-\boldsymbol{W})^{\mathrm{T}}(\boldsymbol{I}-\boldsymbol{W})$。
- 对 $\boldsymbol{M}$ 进行特征值分解，取其最小的 n' 个特征值对应的特征向量，即得到样本集在低维空间中的矩阵 $\boldsymbol{Z}$。

【例 3-7】 利用局部线性嵌入对数据进行降维。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import   datasets,manifold

def load_data():
    '''
    加载用于降维的数据
    :return: 一个元组,依次为训练样本集和样本集的标记
    '''
    iris = datasets.load_iris()              #使用 scikit-learn 自带的 iris 数据集
    return  iris.data,iris.target
def test_LocallyLinearEmbedding(*data):
    '''
    测试 LocallyLinearEmbedding 的用法
    :param data: 可变参数.它是一个元组,这里要求其元素依次为:训练样本集、训练样本的标记
    '''
    X,y = data
    for n in [4,3,2,1]:                      #依次考查降维目标为四维、三维、二维、一维
        lle = manifold.LocallyLinearEmbedding(n_components = n)
        lle.fit(X)
        print('reconstruction_error(n_components = %d) : %s'%
            (n, lle.reconstruction_error_))
def plot_LocallyLinearEmbedding_k(*data):
    '''
    测试 LocallyLinearEmbedding 中 n_neighbors 参数的影响,其中降维至二维
    :param data: 可变参数.它是一个元组,这里要求其元素依次为:训练样本集、训练样本的标记
    :return: None
    '''
    X,y = data
    Ks = [1,5,25,y.size-1]                   #n_neighbors 参数的候选值的集合
    fig = plt.figure()
    for i, k in enumerate(Ks):
        lle = manifold.LocallyLinearEmbedding(n_components = 2,n_neighbors = k)
        X_r = lle.fit_transform(X)           #原始数据集转换到二维
        ax = fig.add_subplot(2,2,i+1)##两行两列,每个单元显示不同 n_neighbors 参数的
LocallyLinearEmbedding 的效果图
        colors = ((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
            (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)#颜色集合,不同标记的样
                                                    #本染不同的颜色
```

```
        for label ,color in zip( np.unique(y),colors):
            position = y == label
            ax.scatter(X_r[position,0],X_r[position,1],label = "target =  %d"
             % label,color = color)
        ax.set_xlabel("X[0]")
        ax.set_ylabel("X[1]")
        ax.legend(loc = "best")
        ax.set_title("k =  %d" % k)
    plt.suptitle("LocallyLinearEmbedding")
    plt.show()
def plot_LocallyLinearEmbedding_k_d1( * data):
    '''
    测试 LocallyLinearEmbedding 中 n_neighbors 参数的影响,其中降维至一维
    :param data: 可变参数.它是一个元组,这里要求其元素依次为:训练样本集、训练样本的标记
    '''
    X,y = data
    Ks = [1,5,25,y.size - 1]                    # n_neighbors 参数的候选值的集合
    fig = plt.figure()
    for i, k in enumerate(Ks):
        lle = manifold.LocallyLinearEmbedding(n_components = 1,n_neighbors = k)
        X_r = lle.fit_transform(X)              # 原始数据集转换到一维
        ax = fig.add_subplot(2,2,i + 1)         # 两行两列,每个单元显示不同 n_neighbors 参数
                                                # 的 LocallyLinearEmbedding 的效果图
                                                # 颜色集合,不同标记的样本染不同的颜色
        colors = ((1,0,0),(0,1,0),(0,0,1),(0.5,0.5,0),(0,0.5,0.5),(0.5,0,0.5),
            (0.4,0.6,0),(0.6,0.4,0),(0,0.6,0.4),(0.5,0.3,0.2),)
        for label ,color in zip( np.unique(y),colors):
            position = y == label
            ax.scatter(X_r[position],np.zeros_like(X_r[position]),
            label = "target =  %d" % label,color = color) # FFFFFF
        ax.set_xlabel("X")
        ax.set_ylabel("Y")
        ax.legend(loc = "best")
        ax.set_title("k =  %d" % k)
    plt.suptitle("LocallyLinearEmbedding")
    plt.show()
if __name__ == '__main__':
    X,y = load_data()                           # 产生用于降维的数据集
    test_LocallyLinearEmbedding(X,y)            # 调用 test_LocallyLinearEmbedding
    plot_LocallyLinearEmbedding_k(X,y)          # 调用 plot_LocallyLinearEmbedding_k
    plot_LocallyLinearEmbedding_k_d1(X,y)       # 调用 plot_LocallyLinearEmbedding_k_d1
```

运行程序,输出如下,效果如图 3-8 和图 3-9 所示。

```
reconstruction_error(n_components = 4) : 7.199368860901911e - 07
reconstruction_error(n_components = 3) : 3.870605052928055e - 07
reconstruction_error(n_components = 2) : 6.641420116916785e - 08
reconstruction_error(n_components = 1) : 1.6515558016659176e - 16
```

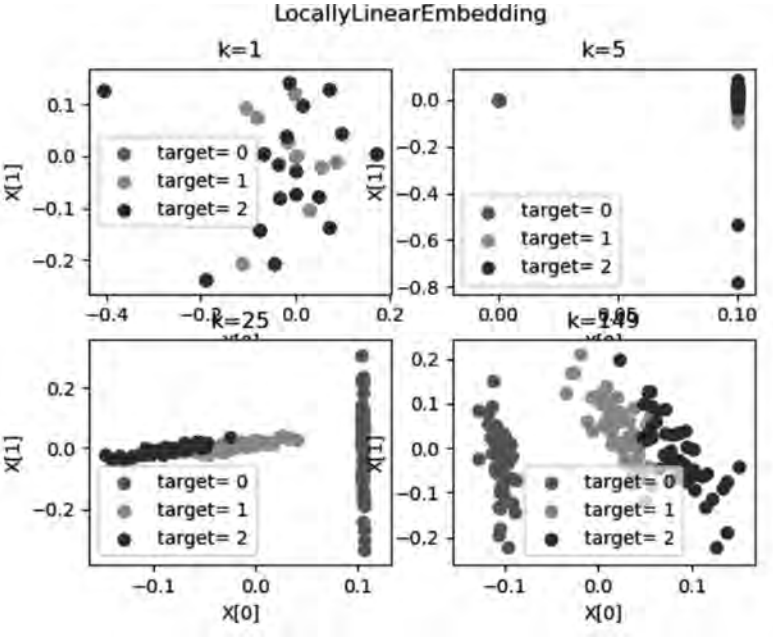

图 3-8 数据 LLE 降维效果

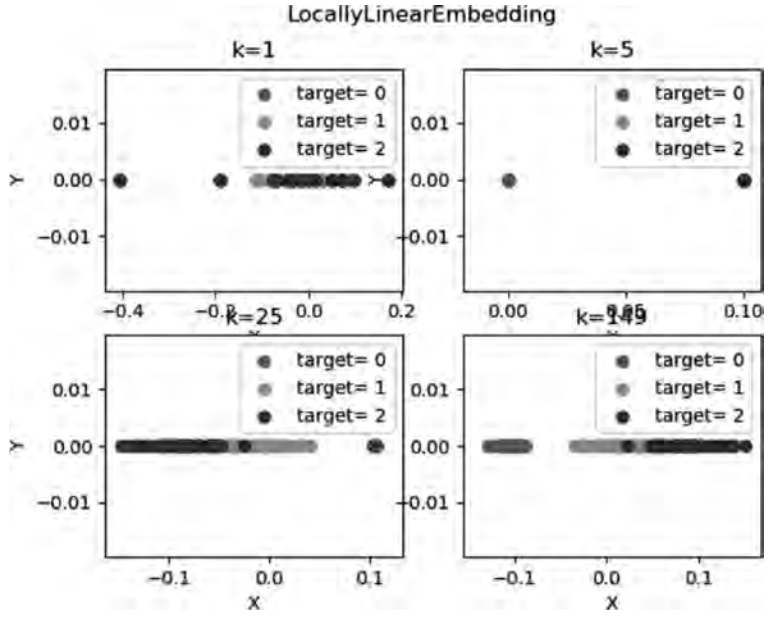

图 3-9 将数据降到一维的效果

3.9 非负矩阵分解

非负矩阵分解(Non-negative Matrix Factorization,NMF)是在矩阵中所有元素均为非负数约束条件之下的矩阵分解方法。其基本思想:给定一个非负矩阵 $\boldsymbol{V}$,NMF 能够找到一

个非负矩阵 $\boldsymbol{W}$ 和一个非负矩阵 $\boldsymbol{H}$，使得矩阵 $\boldsymbol{W}$ 和 $\boldsymbol{H}$ 的乘积近似等于矩阵 $\boldsymbol{V}$ 中的值。

$$\boldsymbol{V}_{n\times m}=\boldsymbol{W}_{n\times k}\times\boldsymbol{H}_{k\times m}$$

其中，$\boldsymbol{W}$ 为基础图像矩阵，相当于从原始矩阵 $\boldsymbol{V}$ 中抽取出来的特征；$\boldsymbol{H}$ 矩阵为系数矩阵。

NMF 广泛应用于图像分析、文本挖掘和语音处理等领域。

最小化 $\boldsymbol{W}$ 矩阵 $\boldsymbol{H}$ 矩阵的乘积和原始矩阵之间的差别，其目标函数为：

$$\operatorname{argmin}\frac{1}{2}\|\boldsymbol{H}-\boldsymbol{WH}\|^2=\frac{1}{2}\sum_{i,j}(x_{ij}-\boldsymbol{W}H_{ij})^2$$

基于 KL 散度的优化目标，其损失函数为：

$$\operatorname{argmin}J(\boldsymbol{W},\boldsymbol{H})=\sum_{i,j}\left(x_{ij}\ln\frac{x_{ij}}{\boldsymbol{W}H_{ij}}-x_{ij}+\boldsymbol{W}H_{ij}\right)$$

【例 3-8】 在 sklearn 封装了 NMF 的实现，可以非常方便地使用，其实现基本和理论部分的实现是一致的，但应注意 sklearn 中输入数据的格式是(samples，features)。

```
#导入必要的编程库
from sklearn.decomposition import NMF
from sklearn.datasets import load_iris
#载入数据
X, _ = load_iris(True)
#最重要的参数是 n_components,alpha,l1_ratio,solver
nmf = NMF(n_components = 2,                    #k value,默认会保留全部特征
          init = None,   #W H 的初始化方法,包括'random' | 'nndsvd'(默认) |  'nndsvda' |
'nndsvdar' | 'custom'.
          solver = 'cd',   #'cd' | 'mu'
          #{'frobenius', 'kullback - leibler', 'itakura - saito'},一般默认就好
          beta_loss = 'frobenius',
          tol = 1e - 4,                                #停止迭代的极限条件
          max_iter = 200,                              #最大迭代次数
          random_state = None,
          alpha = 0.,                                  #正则化参数
          l1_ratio = 0.,                               #正则化参数
          verbose = 0,                                 #冗长模式
          shuffle = False                              #针对"cd solver"
          )
# ------------------ 函数 -------------------------
print('params:', nmf.get_params())                     #获取构造函数参数的值,也可以通过
                                                       #nmf.attr 得到,所以下面会省略这些属性
#下面的 4 个函数很简单,也最核心
nmf.fit(X)
W = nmf.fit_transform(X)
W = nmf.transform(X)
nmf.inverse_transform(W)
# ------------------ 属性 -------------------------
H = nmf.components_                                    #H 矩阵
print('reconstruction_err_', nmf.reconstruction_err_)  #损失函数值
print('n_iter_', nmf.n_iter_)                         #实际迭代次数
```

运行程序，输出如下：

```
params: {'alpha': 0.0, 'beta_loss': 'frobenius', 'init': None, 'l1_ratio': 0.0, 'max_iter': 200,
'n_components': 2, 'random_state': None, 'shuffle': False, 'solver': 'cd', 'tol': 0.0001, 'verbose': 0}
```

```
reconstruction_err_ 3.9480195652425465
n_iter_ 199
```

在以上代码中，各参数的含义为：

- init 参数中，nndsvd（默认）更适用于 sparse factorization，其变体则适用于 dense factorization。
- solver 参数中，如果初始化中产生很多零值，Multiplicative Update（mu）不能很好地更新。所以 mu 一般不和 nndsvd 一起使用，而和其变体 nndsvda、nndsvdar 一起使用。
- solver 参数中，cd 只能优化 Frobenius norm 函数，而 mu 可以更新所有损失函数。

【例 3-9】 一个 NMF 在图像特征提取的应用例子。

```
#导入必要的编程库
from time import time
from numpy.random import RandomState
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_olivetti_faces
from sklearn import decomposition
#设置参数
n_row, n_col = 2, 3
n_components = n_row * n_col
image_shape = (64, 64)
rng = RandomState(0)
#载入 face 数据
dataset = fetch_olivetti_faces('./', True, random_state=rng)
faces = dataset.data
n_samples, n_features = faces.shape
print("Dataset consists of %d faces, features is %s" % (n_samples, n_features))
#显示原始图像
def plot_gallery(title, images, n_col=n_col, n_row=n_row, cmap=plt.cm.gray):
    plt.figure(figsize=(2. * n_col, 2.26 * n_row))
    plt.suptitle(title, size=16)
    for i, comp in enumerate(images):
        plt.subplot(n_row, n_col, i + 1)
        vmax = max(comp.max(), -comp.min())
        #显示压缩后的图像
        plt.imshow(comp.reshape(image_shape), cmap=cmap,
                   interpolation='nearest',
                   vmin=-vmax, vmax=vmax)
        plt.xticks(())
        plt.yticks(())
    plt.subplots_adjust(0.01, 0.05, 0.99, 0.93, 0.04, 0.)

estimators = [
    ('Non-negative components - NMF',
     decomposition.NMF(n_components=n_components, init='nndsvda', tol=5e-3))
]
#绘制输入数据的示例
plot_gallery("First centered Olivetti faces", faces[:n_components])
```

```
#估算并绘制它
for name, estimator in estimators:
    print("Extracting the top %d %s..." % (n_components, name))
    t0 = time()
    data = faces
    estimator.fit(data)
    train_time = (time() - t0)
    print("done in %0.3fs" % train_time)
    components_ = estimator.components_
    print('components_:', components_.shape, '\n**\n', components_)
    plot_gallery('%s - Train time %.1fs' % (name, train_time),
                 components_)
plt.show()
```

运行程序，输出如下，效果如图 3-10 和图 3-11 所示。

```
downloading Olivetti faces from https://ndownloader.figshare.com/files/5976027 to ./
Dataset consists of 400 faces, features is 4096
Extracting the top 6 Non-negative components - NMF...
done in 0.500s
components_: (6, 4096)
**
[[0.         0.         0.         ... 1.89640523 1.78331733 1.68142998]
 [0.95390665 1.02885565 1.09771352 ... 0.25895402 0.31447183 0.32589285]
 [0.02238854 0.01136337 0.06086569 ... 0.13581616 0.14109341 0.15405469]
 [0.18907826 0.26982826 0.38558988 ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.51045833 0.52450599 0.50316775 ... 0.03719877 0.02762018 0.04051619]]
```

图 3-10　降维前的图像

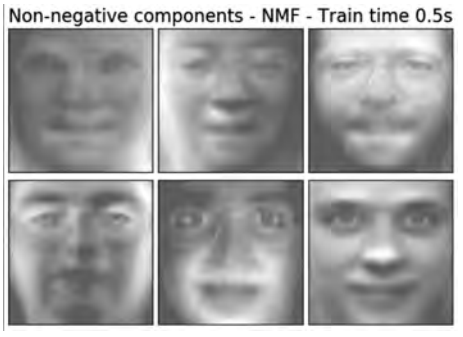

图 3-11　降维后的图像

3.10　小结

数据降维基本原理是将样本点从输入空间通过线性或非线性变换映射到一个低维空间，从而获得一个关于原数据集紧致的低维表示。本章从维度灾难与降维、主成分分析、SVD 降维、核主成分分析（KPCA）降维、多维缩放（MDS）降维、局部线性嵌入（LLE）、非负矩阵分解等多个方面介绍了数据降维相关内容，每个小节都是通过理论、图文、实例相结合

进行数据降维介绍，让读者快速上手利用 Python 解决实际降维问题。

3.11 习题

1. 数据降维，一方面可以解决_____，缓解_____、_____现状，降低复杂度；另一方面可以更好地_____和_____数据。

2. 根据是否考虑和利用数据的监督信息可以划分为_____、_____和_____。

3. 缓解维度灾难的一个重要途径是什么？

4. 什么是 PCA？

5. PCA 降维的准则有几个？分别是什么？