数据降维方法及应用

齐天博

tianboqi.github.io

目录

1　数据降维的基本概念

1.1　数据

1.1.1　什么是数据

数据是对事物通过观测得到的数字性的描述．对于同一个事物，我们可以描述其多种不同的 **特征** (feature)． 例如，对于一个人，我们可以描述其身高、体重、年龄等．这样，每个数据点就可以被写作一个实数组．如果我们把每一个特征作为一个维度，那么这个实数组就可以视作高维空间中的一个点的坐标．我们用 $\boldsymbol{x}^{(i)}$ 表示第 i 个数据点的坐标，并用下标表示其不同的特征，即维度．按照数据科学的传统，我们把这个数组写成行向量的形式，也就是

有一些特征可以是分类的而非数字描述的，这种情况下不太常进行降维．

$$\boldsymbol{x}^{(i)} = (x_1^{(i)}, x_2^{(i)}, \cdots, x_m^{(i)})$$

我们把所有的数据点的坐标纵向排列成一个矩阵，称为 **设计矩阵** (design matrix)

$$\boldsymbol{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_m^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_m^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(n)} & x_2^{(n)} & \cdots & x_m^{(n)} \end{bmatrix}$$

而有的时候，每个数据点还会有一个额外的 **标签** (label)，用 $y^{(i)}$ 表示．它往往代表对于数据点的分类，并不作为数据的一个特征维度．所有数据点的标签也会被排成一个向量，并且按照传统往往是列向量．

这里的 $\top$ 表示转置，也就是把行向量变成列向量．不过标签写成行向量还是列向量往往没有那么严格，因为在计算机中它会被存成一维数组，没有行列之分．

$$\boldsymbol{y} = (y^{(1)}, y^{(2)}, \cdots, y^{(n)})^\top$$

还有两个常用的记号是，我们会把所有的 $\boldsymbol{x}$ 数据点的集合记作 $\mathcal{X}$，把所有 y 的集合记作 $\mathcal{Y}$.

1.1.2　数据所在的空间*

我们前面提到，数据就是一组在 $\mathbb{R}^m$ 空间中的点．在对这组点进行降维时，我们往往需要使用数据点之间的距离．定义了距离的空间称为 **度量空间**．最常见的距离度量即为满足勾股定理的度量，称为欧几里得度量或 L^2 度量．即

$$d(\boldsymbol{p}, \boldsymbol{q}) = \sqrt{\sum_{i=1}^{m} (p_i - q_i)^2}$$

不过我们也完全可以使用其他的度量，例如 L^p 度量．在很多情况下，我们甚至可以使用在数学上不严格构成度量的距离定义，例如余弦相似度等．

从度量的定义中我们可以发现一个问题——当使用的单位不同时，数据点之间的距离关系可能会有很大不同．例如要描述一个人的身高体重，如果我们使用 (m, kg) 作为单位，那么体重坐标将远大于身高坐标，因此数据点之间的距离将主要由体重决定；而如果我们使用 (cm, lb) 作为单位，那么身高和体重的数值会在同一个数量级上，因此二者对距离的影响相当．一般来说，我们会对数据的每个维度分别进行一些线性变换（等价于单位变换），使得数据整体在各个维度的数值在相似的范围内，这称为数据的标准化或归一化．

归一化和标准化实际上代表两种不同的方式．所谓 **标准化** (standardization) 是指将数据的每个维度各自的均值变为 0、方差变为 1，即

$$x_i \leftarrow \frac{x_i - \mu_i}{\sigma_i}$$

其中 μ_i 为第 i 个维度的均值，σ_i 为第 i 个维度的方差．

而 **最大值-最小值归一化** (min-max normalization) 是指将每个维度各自都缩放到区间 [0,1] 中，即

$$x_i \leftarrow \frac{x_i - \min(x_i)}{\max(x_i) - \min(x_i)}$$

标准化和归一化都很常用，并且往往可以相互替代．一般来说，对于正态分布的变量，人们更习惯使用标准化．有些时候，数据的预处理要求数据的中心位于零，那么此时标准化可以自动满足．而如果使用归一化，则需要将数据进一步变换到 [-1,1] 中．

1.1.3　示例数据

我们在这个笔记中使用 Python 语言，并主要使用 scikit-learn 库中著名的玩具数据集——鸢尾花（Iris）数据集． 这个数据集收录了三种不同的鸢尾花个体的萼片及花瓣的长度和宽度．我们通过下面的代码加载这个数据集．

这三种鸢尾花分别是 *Iris setosa*、*I. virginica* 和 *I. versicolor*.

```
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target
```

这样，Iris 数据集的特征 $\boldsymbol{X}$ 就被存储在了 `X` 这个数组中．这是一个 150×4 的数组，每一行表示一个鸢尾花数据点，总共 150 个数据点．而四列分别表示萼片长度、萼片宽度、花瓣长度和花瓣宽度（cm）．下面显示了前 10 个数据点．

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4],
       [4.6, 3.4, 1.4, 0.3],
       [5. , 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.1]])
```

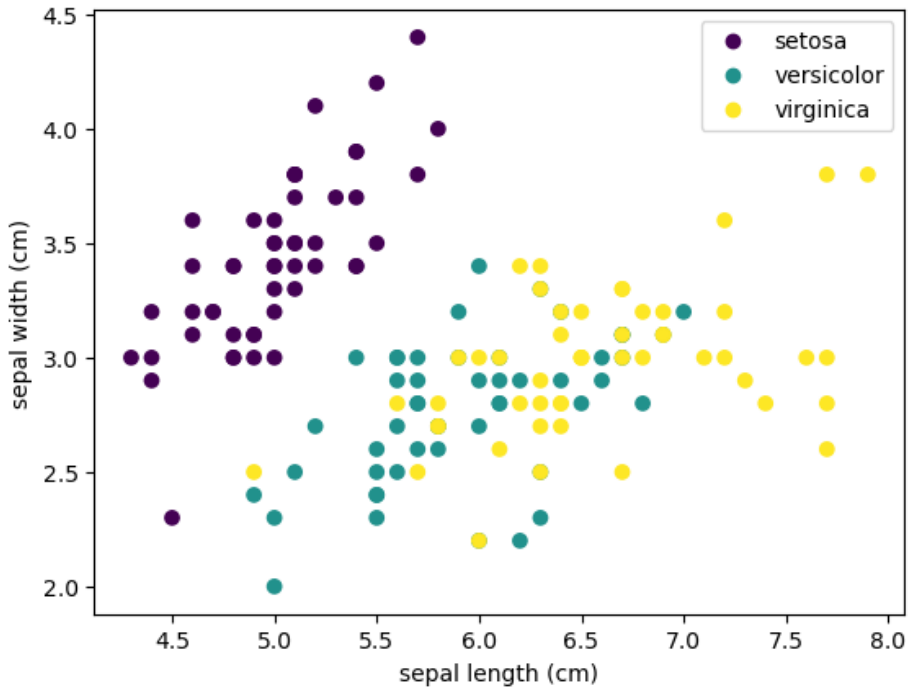
而鸢尾花的种类——也就是标签 $\boldsymbol{y}$ ——则被存在 `y` 中．这是一个 150 维数组，数字 0、1、2 分别表示三种不同的鸢尾花，它们的名字存在 `iris.target_names` 中．整个数组如下．

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

下面我们以萼片的长度和宽度为例，画出这个数据集.

```
import matplotlib.pyplot as plt

_, ax = plt.subplots()
scatter = ax.scatter(X[:, 0], X[:, 1], c=y)
ax.set(xlabel=iris.feature_names[0], ylabel=iris.feature_names[1])
_ = ax.legend(scatter.legend_elements()[0], iris.target_names)
```

输出如下.



而数据的归一化和标准化可以通过 `sklearn.preprocessing` 库里的 `StandardScaler` 或 `MinMaxScaler` 类来实现. 例如下面是对数据 `X` 进行标准化的代码.

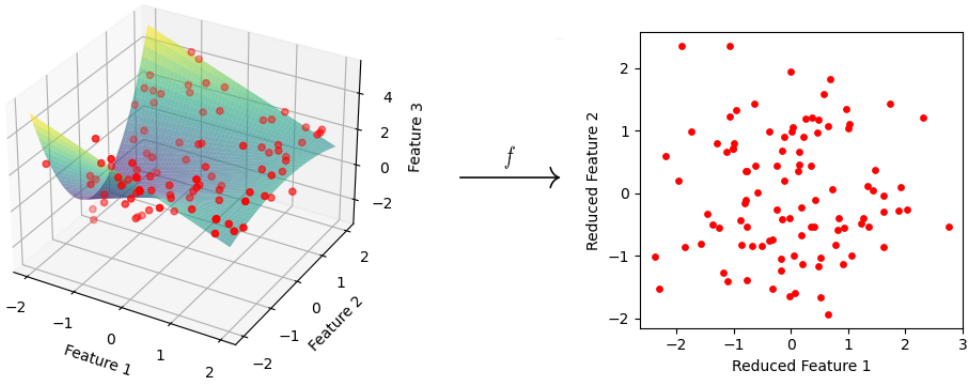
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X)
X_standardized = scaler.transform(X)
```

这样，`X_standardized` 就是标准化后的数据数组.

1.2　降维

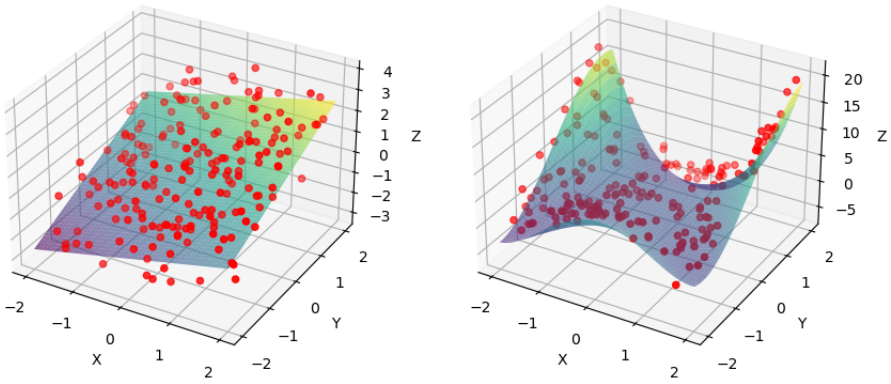
1.2.1　什么是降维

在大数据时代，我们常常会接触到很高维的数据. 有时，由于数据的不同维度之间的非独立性，抑或是一些其他我们后面会提到的原因，我们会希望把数据降低至一个较低的维度，这就称为数据降维. 严格地说，若数据是存在于 $\mathbb{R}^n$ 中的点，那么降维则是找到一个函数 $f : \mathbb{R}^n \to \mathbb{R}^p$，把数据映为低维空间 $\mathbb{R}^p$ 中的点. 当然，我们并不希望找到一个很随意的降维函数. 我们往往希望邻近的点在降维之后仍然能保持邻近. 这在直观上有点类似于在高维的数据空间中找到一个低维子空间（也就是一个“超曲面”），将数据点“投影” 到这个子空间以后，再将子空间“展平”，如下图所示.



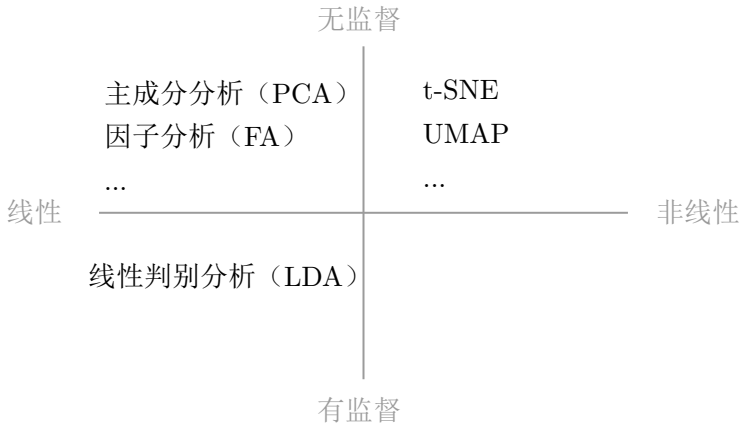
降维算法可以被分为几大类．首先，根据降维函数 f 是否是一个线性函数，或者直观地说，根据降维后的空间是否是平直的，我们可以把降维方法分为 **线性** (linear) 和 **非线性** (nonlinear) 降维，如下图所示．其中非线性降维也被称为 **流形学习** (manifold learning)．线性降维往往较为简单，并且可解释性高．但有时数据的不同维度之间的关系并不是线性的，也就是说数据点并不位于一个平直空间附近，此时非线性降维则更为合适．

流形是微分几何的概念，不严格地说，它是指弯曲的空间，例如曲线和曲面．



而另一种分类方式则与数据点的标签有关．有时，我们希望降维可以顺便帮助我们将不同标签的数据点分开，这种降维方法称为 **有监督** (supervised) 降维．反正，如果不考虑数据点的标签，则称为 **无监督** (unsupervised) 降维．

综合以上两种分类方式，我们可以将降维方法分类四大类．不过其实不太存在通用的有监督非线性降维方式，所以其实只有三大类．我们下面将按照这种分类方式逐个讲解常见的降维算法．



1.2.2　数据降维的意义

首先，数据降维是数据可视化的一种常用方式．我们常常会接触到非常高维的数据，甚至可以高达几万维．然而，若想要直观地画出这些数据，我们只能把它们画在二维的纸或者屏幕上，或者至多画成一个三维立体的图．这就需要我们将数据降至二维或三维．

第二，数据降维是数据预处理的一个常用步骤，因为它可以极大地降低后续计算量．我们拿到的数据往往会在后续进行一些处理、建模等分析过程．而处理高维数据意味着更大的计算量，也就是更高的内存使用和更长的时间．同时，高维空间的数据较为稀疏，且其分布存在着一些反常识的现象，会降低机器学习算法的有效性，这称为 **维数灾难**．降维算法一定程度上可以避免维数灾难．因此，对于高维数据来说，降维几乎是预处理中的一个必需步骤．例如在机器学习中的词嵌入、特征提取等步骤实际上就是一种降维过程．

第三，数据降维可以用于分析数据的内部结构．我们收集的高维数据的不同维度之间往往并不是完全独立的，而这种维度间的制约关系会导致数据存在于一个相对低维的空间中．找到这个低维空间可以帮助我们分析不同维度之间的关联性，或推断影响这些维度的隐藏变量．

2　无监督线性降维方法

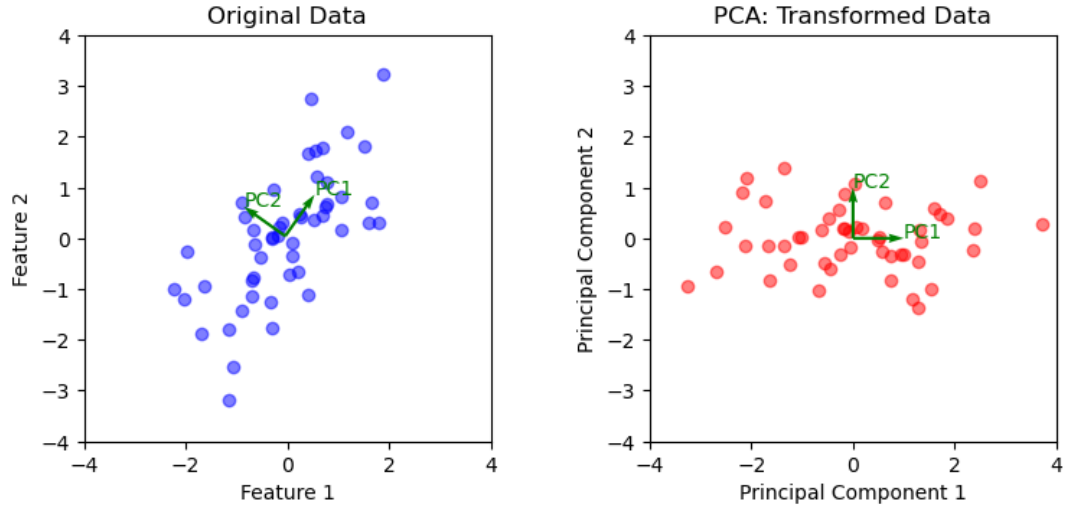
我们在这里只介绍三种最常见的无监督线性降维方法——主成分分析 (PCA)、因子分析 (FA) 和独立成分分析 (ICA)．其他一些较为少见的方法如非负矩阵分解 (NMF) 和偏最小二乘 (PLS) 等在此不进行讲解．

2.1　主成分分析

下面我们首先来介绍 **主成分分析** (principal component analysis, **PCA**)，这是一种最常见的无监督线性降维方法．

2.1.1　主成分分析的统计学原理

现在假设我们有一组 m 维数据，总共有 n 个数据点．PCA 想要回答的问题是：在这 m 维空间中，哪个方向最能体现数据的差异？例如下图中，左图的数据点在左下-右上的方向上差异最大，这个方向被称为第一主成分（PC1）．找到第一主成分后，我们继续问，在垂直于 PC1 ——也就是与 PC1 不相关——的所有方向上，哪个方向数据的差异最大？例如下图中就是左上-右下的方向，这被称为第二主成分（PC2）．以此类推，我们最多能找到 m 个主成分．我们可以用这些主成分重新表示原始数据，也就是把数据投影到这些主成分上．这就是主成分分析．



需要指出的是，上图中我们将二维数据通过 PCA 变成了另一种表示方式下的二维数据，这是为了方便表示出 PCA 所寻找的主成分的方向．但是，我们可以只选择前几个主成分，这样变换后的数据低于原始数据．从几何上来看，这就相当于把数据点正交投影到了保留的主成分所张成的低维子空间中，从而实现降维．

以上是 PCA 所希望实现的目标，而下面我们来看一下实现 PCA 的数学原理．从上面的例子可以看出，PCA 关注数据各个维度之间的线性关系，而体现这一点的统计概念是数据的 **协方差**．对于数据的两个维度 x_i 和 x_j，每个维度有 n 次观察，我们定义这两个维度的协方差

$$\sigma_{ij} = \frac{1}{n-1}\sum_{k=1}^{n}(x_i^{(k)} - \bar{x}_i)(x_j^{(k)} - \bar{x}_j)$$

其中 $\bar{x}_i$ 是第 i 个维度的均值． 协方差体现了数据这两个维度的变化趋势是否相同，类似于相关系数．并且很容易发现，一个分量和它自身的协方差就是其方差 $\sigma_{ii} = \sigma_i^2$．我们可

实际上，协方差 σ_{ij} 和相关系数 r_{ij} 的关系是

$$r_{ij} = \frac{\sigma_{ij}}{\sigma_i \sigma_j}$$

以把协方差排列成一个矩阵，称为协方差矩阵

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \cdots & \sigma_{1m} \\ \sigma_{21} & \sigma_{22} & \cdots & \sigma_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{m1} & \sigma_{m2} & \cdots & \sigma_{mm} \end{bmatrix}$$

这是一个对称矩阵，因为显然有 $\sigma_{ij} = \sigma_{ji}$. 现在我们假设数据是中心化的，即所有的数据点的均值位于原点处. 如果使用数据的设计矩阵 $\boldsymbol{X}$，那么协方差矩阵可以用矩阵乘法写作

对于非中心化的数据，我们一般首先会对其做中心化.

$$\boldsymbol{\Sigma} = \frac{1}{n-1}\boldsymbol{X}^\top \boldsymbol{X}$$

从线性代数的角度来说，PCA 就是希望找到一组新的标准正交基，使得每个分量代表其递降的主成分. 我们将这个过程的基变换矩阵写作 $\boldsymbol{P}$，则 $\boldsymbol{P}$ 是一个正交矩阵，且其列向量代表了新基（主成分）在原基观测变量下的坐标，称为 **载荷** (loading). 设新基下的数据矩阵为 $\boldsymbol{X}'$，则有 $\boldsymbol{X} = \boldsymbol{P}\boldsymbol{X}'$. 我们把这个变换带进上式，得到

正交矩阵也就是保持新的基仍然为标准正交基的矩阵.

$$\boldsymbol{\Sigma} = \frac{1}{n-1}\boldsymbol{P}^\top \boldsymbol{X}'^\top \boldsymbol{X}' \boldsymbol{P} = \boldsymbol{P}^\top \boldsymbol{\Sigma}' \boldsymbol{P}$$

其中 $\boldsymbol{\Sigma}'$ 是基变换后的协方差矩阵. 由上面对主成分的定义，我们希望 $\boldsymbol{\Sigma}'$ 中，方差项（也就是对角项）是递减的. 并且由于变换后的各个主成分之间是不相关的，也就是任意不同的两个分量的协方差均为零，所以这个矩阵的非对角元为 0. 这也就是线性代数中重要的相似对角化的过程. 此时 $\boldsymbol{\Sigma}'$ 形如

$$\boldsymbol{\Sigma}' = \begin{bmatrix} \sigma'_{11} & & \\ & \ddots & \\ & & \sigma'_{mm} \end{bmatrix} = \begin{bmatrix} \sigma'^2_{1} & & \\ & \ddots & \\ & & \sigma'^2_{m} \end{bmatrix}$$

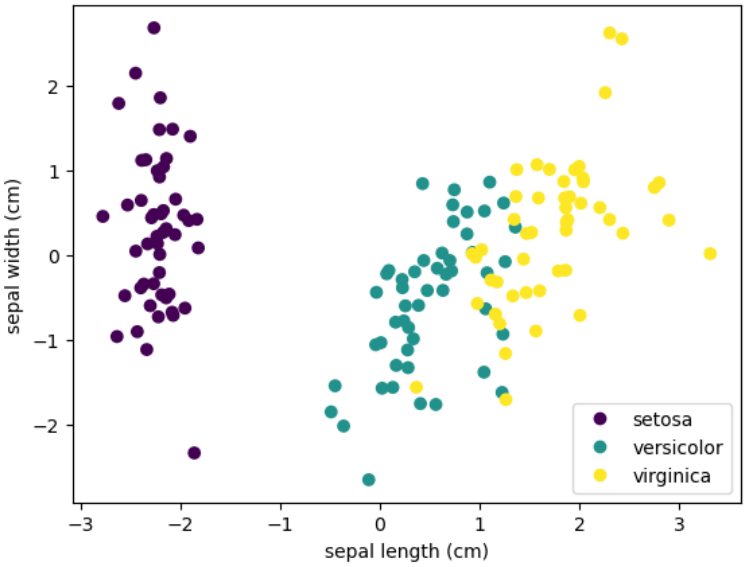
也就是说，PCA 就是在对协方差矩阵 $\boldsymbol{\Sigma}$ 进行相似对角化，其对角元就是协方差矩阵的特征值，即各个主成分方向上的方差. 而基变换矩阵 $\boldsymbol{P}$ 的列向量代表了主成分. 换句话说，PC1 就是最大的特征值所对应的特征向量，该特征值对应了 PC1 方向的方差；PC2 是第二大的特征值所对应的特征向量，该特征值对应了 PC2 方向的方差，以此类推.

2.1.2　主成分分析的代码实现

我们下面使用 sklearn 库来简单地实现一下 Iris 数据集的主成分分析. PCA 要求数据的均值为零，因此我们这里使用标准化的数据 `X_standardized`. 我们下面可以直接调包进行实现. sklearn 的编程逻辑通常都是相同的，即建立示例，拟合数据，再对数据进行计算. 例如，下面的代码把 4 维的 Iris 数据集降维为 2 维.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)                # PCA instance
pca.fit(X_standardized)                  # fit the data
X_pca = pca.transform(X_standardized)    # transform onto new bases
```

此时 `X_pca` 是一个二维数据矩阵，画出来如下图所示. 我们也可以不传入 `n_components` 参数，直接以 `PCA()` 进行实例化. 这样，PCA 算法会保留所有的主成分，也就是说 `X_pca` 会是一个四维数组. 此时如果我们只取结果的前两维，会得到相同的结果.



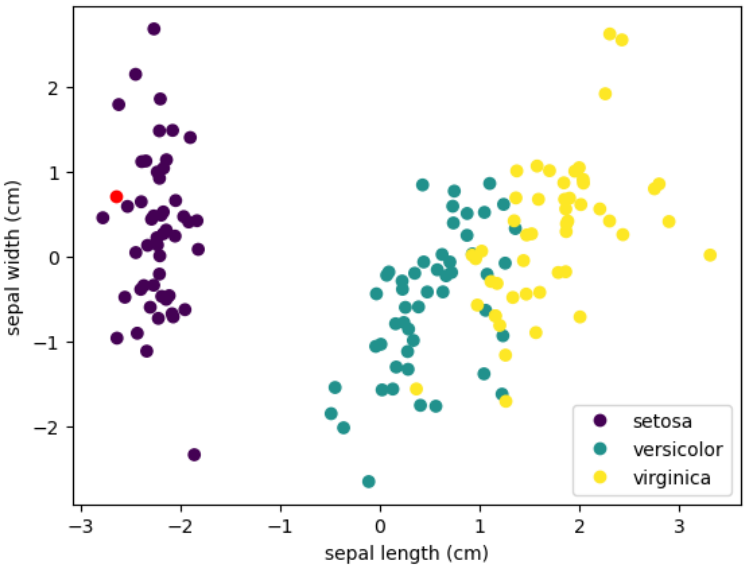
注意 PCA 是一种非监督降维算法，也就是说在计算主成分时我们完全没有考虑数据点的标签．三个不同类别的数据点在此相互分开完全是由于数据自身的结构导致的．

另外，此时的 `pca` 实例已经是一个被训练过后的算法，它的 `transform()` 方法不仅可以降维 `X_standardized`，也可以按照同一个降维函数来降维任何数据点． 例如，若我们有一个新的数据点 `X_new`，我们可以把它按照同样的标准化函数处理后进行降维，整合进上面的图里．

注意这里的 `scaler()` 也是用之前的数据训练的．

```
x_new = np.array([[4.5, 2.9, 1.6, 0.1]])
x_new_pca = pca.transform(scaler.transform(x_new))
```

将这个点整合到所有的数据点中，如下图的红点所示．可以看出，这个数据点更有可能是 *I. setosa*．另外，需要强调的是，我们上面的操作中完全是使用原始数据集对 PCA 算法进行的训练，这与将这个点整合进原数据集中再进行 PCA 降维的结果是不一样的．



2.1.3　如何选择主成分的数目

对于一个降维算法，我们总是需要人工选择降维后维度的数目，在 PCA 算法中也就

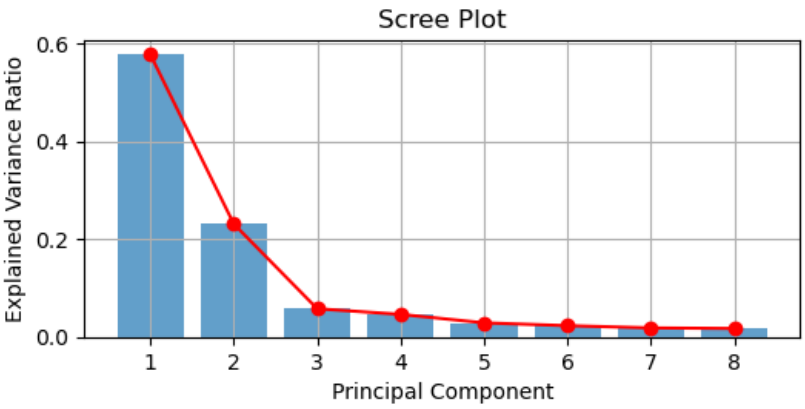
是主成分的数目．有时，维度的数目是完全定好的，例如若要把数据画在平面上，那么我们就必须将数据降为二维．

而有时，我们只是想要降低数据量，而基本保持数据的原有结构．这时，我们通常会要求 PCA 保留原有数据的一定方差．我们前面说过，对角化的协方差矩阵的对角元 σ'_{ii} 就代表各个主成分方向的方差，也就是各个主成分所解释的数据方差的大小．m 个主成分共同解释了数据的所有方差．由此我们就可以计算每个主成分分别解释了数据中多少比例的方差．在 sklearn 中，这被存在PCA 实例的 `explained_variance_ratio_` 属性中．我们可以看前几个主成分满足我们对所保留的方差的比例的要求，来选择主成分的数目．

例如，在上面的例子中，我们可以把 PCA 算法实例化为 `PCA = pca()`．在不传入 `n_components` 参数时，上面的代码会自动计算出所有的主成分，也就是保持维数不变．此时 `pca.explained_variance_ratio_` 会记录下所有主成分方向所解释的方差比例．输出为一个数组 `array([0.92461872, 0.05306648, 0.01710261, 0.00521218])`．若我们希望保留 90% 的方差，那么可以只取第一个主成分；而若要保留 95% 的方差，则可以取前两个主成分．

在上面的方法中，PCA 保留的方差比例是人为规定的．若不想人为设置，则我们可以观察从哪个主成分开始，所解释的方差大幅降低．我们将各个主成分方向的特征值，也就是所解释的方差画成下图，称为 **碎石图** (scree plot)．可以看到，第三个主成分处是一个“拐点”，也就是从 PC3 开始，所解释的方差大幅降低．因此，我们可以只保留前两个主成分．

我们画的不是 Iris 数据集的碎石图，因为 Iris 数据集的维度太低，拐点不明显．



2.2　因子分析

我们下面来讨论一个与 PCA 非常类似的降维方法，称为 **因子分析** (factor analysis, **FA**)．实际上，FA 也分为几种不同的类型，而我们这里只讲解最常用的一种，称为探索性因子分析 (exploratory factor analysis, EFA)．我们先来讲解 FA 的原理和实现，最后再去讨论它和 PCA 的异同．

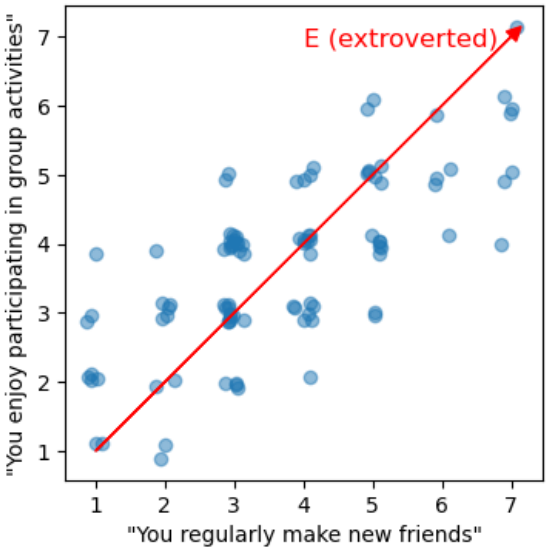
2.2.1　观测变量与隐藏变量

我们收集到的高维数据的各个维度之间经常会存在一定的相关性，其中常见的原因之一是它们体现了某些相同或相似的性质．也就是说，存在着一些我们没有观测到的 **隐藏变量** (latent variables)，也称为 **公共因子**，简称因子 (factors)．而我们的数据直接记录的变量，称为 **观测变量** (observed variables)，会受到这些隐藏变量的影响．因子分析的目的就是通过观测变量之间的相关性，来找到可能存在的隐藏变量．

下面我们以著名的 MBTI 性格测试为例来解释一下隐藏变量是什么． MBTI 性格测试是通过一个包含上百道题目的量表，将人的性格在四个维度上进行区分．为了使得该测试的有较高的信度，会有许多题目指向的是同一个性格维度．例如下面这两道题目的相关性往往极高，这是因为它们都是由受试者的内向/外向性决定的，这就是一个隐藏变量．当然，我们这里只展示了两个观测变量所体现的一个隐藏变量．实际上，我们可以有很多的观测变量和很多的隐藏变量．一个观测变量也可以由几个隐藏变量共同影响．

实际上，因子分析就起源于心理学．

这里的点是离散的是因为 MBTI 使用的是七点量表，这与因子分析无关．



2.2.2　因子分析的统计学原理

下面我们来用数学语言来写出因子分析的基本假设．设有 r 个隐藏变量，称为 f_1 到 f_r，那么我们观测到的每个变量 x_i 可以写作隐藏变量的线性组合

$$x_i = \sum_{j=1}^{r} \ell_{ij} f_j + \epsilon_i$$

其中 ℓ_{ij} 表示隐藏变量对观测变量的影响大小，仍然称为载荷，而 ϵ 是随机误差，也称为**特殊因子**．考虑到我们有多个数据点，则对整个数据可以用矩阵写作

这里的数据仍然是已经中心化了的．

$$\boldsymbol{X} = \boldsymbol{F}\boldsymbol{L} + \boldsymbol{\epsilon}$$

其中 $\boldsymbol{F} \in \mathbb{R}^{n\times r}$ 是因子矩阵，每一行表示每个数据点在各个隐藏变量上的值； $\boldsymbol{L} \in \mathbb{R}^{r\times m}$ 是载荷矩阵，其每一行表示每个隐藏变量在观测变量上的载荷．而 $\boldsymbol{\epsilon} \in \mathbb{R}^{n\times m}$ 是误差矩阵．

所以，因子分析的目的就是估计矩阵 $\boldsymbol{L}$ 和 $\boldsymbol{F}$．不过，这种矩阵分解方式有无数种．为了使得分析出的因子有意义，因子分析做了以下假设

这里的记号有一点多重含义，$\boldsymbol{X}$ 和 $\boldsymbol{F}$ 是由样本构成的矩阵，但是我们写形如 $\mathrm{Cov}(\boldsymbol{X})$ 的时候把它们看作随机变量．

1. 因子之间是无关的，且因子是归一化的，即 $\mathrm{Cov}(\boldsymbol{F}) = \boldsymbol{I}$；
2. 每个因子都是中心化的，即 $\mathbb{E}(\boldsymbol{F}) = \boldsymbol{0}$；
3. 误差之间是无关的，即 $\mathrm{Cov}(\boldsymbol{\epsilon})$ 为对角矩阵，记作 $\boldsymbol{\Psi}$；
4. 因子和误差之间都是无关的．

在这个假设下，我们可以进一步来分析数据间的相关性．如果我们仍然把数据的协方差记做 $\boldsymbol{\Sigma}$，则有

$$\boldsymbol{\Sigma} = \mathrm{Cov}(\boldsymbol{F}\boldsymbol{L} + \boldsymbol{\epsilon}) = \mathrm{Cov}(\boldsymbol{F}\boldsymbol{L}) + \mathrm{Cov}(\boldsymbol{\epsilon})$$

注意到 $\boldsymbol{F}$ 是随机变量——它是在用因子表示出的数据，而数据是随机变量．而 $\boldsymbol{L}$ 并不是随机变量，它是因子的载荷，而因子虽然是未知的但并不是随机变量．因此我们可以把 $\boldsymbol{L}$ 从 Cov 中拿出来．很容易得到

在一维随机变量中有 $\mathrm{Var}(cX) = c^2\mathrm{Var}(X)$．在多维里这个平方变成了一个二次型．

$$\boldsymbol{\Sigma} = \boldsymbol{L}^\top \mathrm{Cov}(\boldsymbol{F})\boldsymbol{L} + \mathrm{Cov}(\boldsymbol{\epsilon})$$

由我们的假设 $\mathrm{Cov}(\boldsymbol{F}) = \boldsymbol{I}$，$\mathrm{Cov}(\boldsymbol{\epsilon}) = \boldsymbol{\Psi}$．最终我们得到对协方差的分解式

$$\boldsymbol{\Sigma} = \boldsymbol{L}^\top \boldsymbol{L} + \boldsymbol{\Psi}$$

其中 $\boldsymbol{L}^\top \boldsymbol{L}$ 代表了隐藏变量所解释的协方差的部分，其对角元称为方差的 **共同部分** (communality)．而剩余的部分 $\boldsymbol{\Psi}$，也就是被归为误差的部分，其对角元称为 **独特部分** (uniqueness)．

因子分析并没有闭式解，因此人们发展了几种不同的方式来求解．其中最常用的一种称为 **主轴因子法** (principal axis factoring, PAF)，又称为 **共同成分分析** (common factor analysis)．除此之外还有 **极大似然法** 等．这两种求解方式都有些复杂，我们在此不具体讨论．

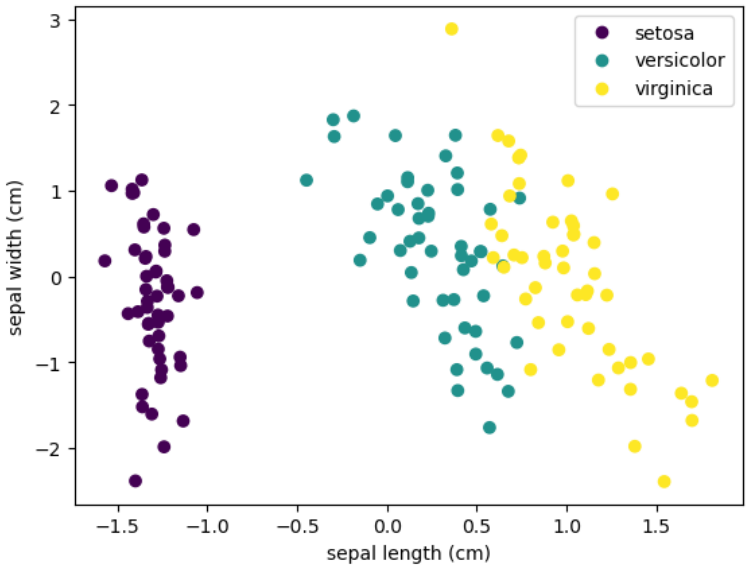
2.2.3　因子分析的代码实现

下面我们来分析一下鸢尾花数据集中可能存在的隐藏变量——例如，鸢尾花的花瓣和萼片的某些长度和宽度的特征是否共同受到某些隐藏变量（例如基因、环境等）的影响？

因子分析可以用和前面的 PCA 和因子分析相同的语法来实现．我们在下面使用一种新的方法 `fit_transform()`，这样可以用一行代码代替原来训练和降维的两行代码．这个方法在 PCA 和等其他算法在也可以使用．

```
from sklearn.decomposition import FactorAnalysis

fa = FactorAnalysis(n_components=2)             # FA instance
X_fa = fa.fit_transform(X_standardized)     # fit and transform the data
```

因子分析的结果如下图所示．可以看到，因子分析得到的结果与 PCA 有所不同．我们下面马上就要讨论因子分析和 PCA 的异同．



2.2.4　因子分析与 PCA 的异同

因子分析与 PCA 都是线性降维算法，并且考虑的都是维度之间的相关性，二者似乎十分类似．但实际上，二者的用途十分不同．下面我们就来具体讨论一下．

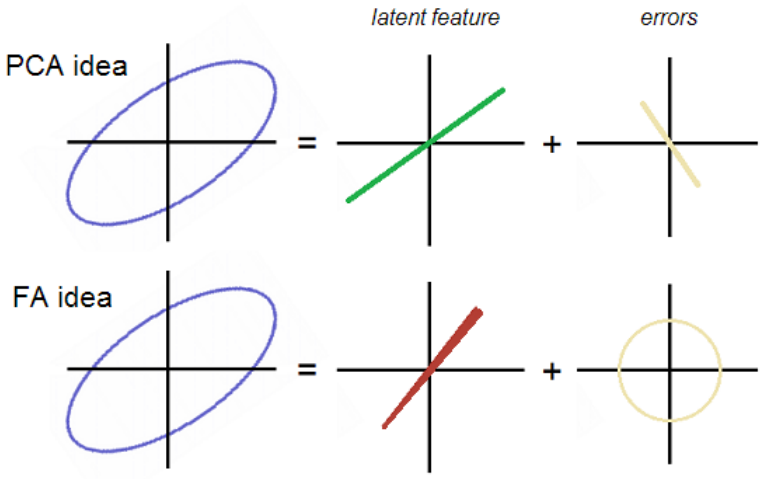
首先，从思想上说，PCA 的唯一目的就是降维，它将数据投影到一个低维子空间，使得投影后数据的的结构——体现在方差上——被最大程度地保留．除此之外，PCA 不关心得到的主成分有什么意义，或者它们与观测变量之间有什么逻辑关系．而因子分析则不同．因子分析则试图建立一个模型，用低维的隐藏变量去解释观测变量中的相关性．并且因子分析希望得到的隐藏变量是有意义的．观测变量是由隐藏变量由线性组合得出的，或者说隐藏变量导致了观测变量，而不是相反．

从使用过程来说，二者的一个重要差异是 PCA 不需要事先指定主成分的数目，但因子分析的结果受到因子数目的影响．在使用 PCA 时，我们可以一下求出所有的主成分，再根据需求取其中的前几个．这和直接求前几个主成分得到的结果是一样的．但在使用因子分析时，由于求解方式的不同，若我们指定了不同的因子数目，那么我们得到的结果也会是不同的．因此，若改变了因子的数目，我们需要重新做因子分析．一般来说，我们可以根据我们所研究的问题而决定因子的数目——例如，在 MBTI 性格测试的例子中，我们的量表希望把性格分为几个维度？在没有先验知识的情况下，我们可以分析方差的公共部分，按照与 PCA 相似的方式找到合适的因子数目．

与此有关的另一个差异是，PCA 得到的主成分是有顺序的，而因子分析得到的因子是没有顺序的．实际上，因子之间不但没有顺序，甚至解都不是唯一的．这是因为因子分析只关心因子线性组合能否解释观测变量，因此张成同一个子空间的一组标准正交基都是等价的．因此，我们可以随意地用正交矩阵去旋转载荷矩阵，这称为 **因子旋转**．其中最常用的一种旋转方式称为方差最大化旋转 (**Varimax**)，它使得因子的方向尽量与某个观测变量相一致，也就是使得每个因子尽量影响较少的观测变量，从而简化模型．

最后，从分析结果来说，因子分析和 PCA 也有重要的不同．因子分析在意的是数据的相关性，而 PCA 在意的是数据的方差．这使得二者得到的低维变量的方向是不同的．换一种方式理解，PCA 中的误差是垂直于所有主成分的，而因子分析的误差是正态分布的．这使得去除了误差的部分的相关性有所差异．这可以从下图中看出．

本图来自 StackExchange 论坛．



3　有监督线性降维方法

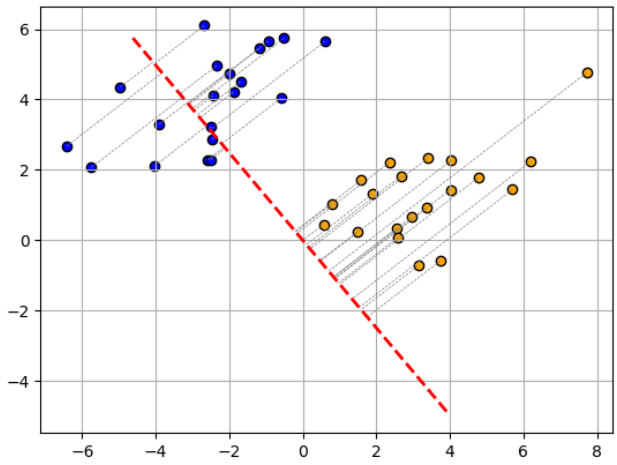
有监督的线性降维方法几乎只存在一种常用的方法——**线性判别分析** (linear discriminant analysis, **LDA**).

3.1　线性判别分析

线性判别分析与 PCA 和因子分析类似，都是线性降维算法．但是 LDA 算法处理的是带有分类标签的数据．它降维时考虑的不再是数据的方差或者相关性，而是将不同的类别之间分开．

3.1.1　双类别 LDA

我们先来看一种较为简单的情况，即只有两个类别，此时的 LDA 也称为 **Fisher 判别分析**．我们的目标是找到一个维度，使得两个类别的数据点向这个维度的投影分得最开——也就是投影后同一个类别的数据点尽量接近，而不同类别的数据点尽量远离，如下图所示．



下面我们来用数学语言描述 LDA 的目的．设两个类别的样本量相等，且均值分别位于 $\boldsymbol{\mu}_0$ 和 $\boldsymbol{\mu}_1$，协方差矩阵为 $\boldsymbol{\Sigma}_0$ 和 $\boldsymbol{\Sigma}_1$．若用 $\boldsymbol{w}$ 表示所降到的一维方向的单位向量，那么每个数据点 $\boldsymbol{x}$ 向这个维度的投影值为 $\boldsymbol{w}^\top\boldsymbol{x}$．则很容易推出，两个类别的均值点向直线的投影值分别为 $\mu_0' = \boldsymbol{w}^\top\boldsymbol{\mu}_0$ 和 $\mu_1' = \boldsymbol{w}^\top\boldsymbol{\mu}_1$．那么我们可以用两个均值的差异的平方来表示类间差异（平方是为了和下面的类内差异保持量纲一致），称为类间 **散度**

$$\text{类间散度} = (\mu_0' - \mu_1')^2 = (\boldsymbol{w}^\top\boldsymbol{\mu}_0 - \boldsymbol{w}^\top\boldsymbol{\mu}_1)^2$$

同时，我们还可以推出两个类别的数据点的投影值的方差分别为 $\sigma_0'^2 = \boldsymbol{w}^\top\boldsymbol{\Sigma}_0\boldsymbol{w}$，$\sigma_1'^2 = \boldsymbol{w}^\top\boldsymbol{\Sigma}_1\boldsymbol{w}$．那么我们可以用投影后的总类内方差作为类内散度

$$\text{类内散度} = \sigma_0'^2 + \sigma_1'^2 = \boldsymbol{w}^\top\boldsymbol{\Sigma}_0\boldsymbol{w} + \boldsymbol{w}^\top\boldsymbol{\Sigma}_1\boldsymbol{w}$$

那么 LDA 的目标就应该是最大化类间散度与类内散度的比值．我们一般把这种需要最大化的函数称为 **目标函数** (objective function)，记作 $J(\boldsymbol{w})$．这里我们的目标函数为

$$J(\boldsymbol{w}) = \frac{\text{类间散度}}{\text{类内散度}} = \frac{(\boldsymbol{w}^\top\boldsymbol{\mu}_0 - \boldsymbol{w}^\top\boldsymbol{\mu}_1)^2}{\boldsymbol{w}^\top\boldsymbol{\Sigma}_0\boldsymbol{w} + \boldsymbol{w}^\top\boldsymbol{\Sigma}_1\boldsymbol{w}}$$

很多地方没有写要求样本量相等．但实际上这里推导的是一种特殊情况，是需要样本量相等的．样本量不等的情况可以见下一节的推广．另外，LDA 作为分类算法时，需要数据的每个类别满足正态分布，且协方差矩阵相同．但 LDA 只作为降维算法时，这两个要求可以不满足．

有的时候我们需要最小化一个函数，这时的函数也可以叫目标函数，或者可以叫做 **损失函数** (loss function)．

最大化上式的 $\boldsymbol{w}$ 即为所求维度的方向．不过为了方便后面推广，我们把上式写成另一种表示方式．我们定义类间散度矩阵 (between-class scatter matrix)

$$\boldsymbol{S}_b = (\boldsymbol{\mu}_0 - \boldsymbol{\mu}_1)(\boldsymbol{\mu}_0 - \boldsymbol{\mu}_1)^\top$$

和类内散度矩阵 (within-class scatter matrix)

$$\boldsymbol{S}_w = \boldsymbol{\Sigma}_0 + \boldsymbol{\Sigma}_1$$

那么目标函数 $J(\boldsymbol{w})$ 可以被改写为

$$J(\boldsymbol{w}) = \frac{\boldsymbol{w}^\top \boldsymbol{S}_b \boldsymbol{w}}{\boldsymbol{w}^\top \boldsymbol{S}_w \boldsymbol{w}}$$

这称为 $\boldsymbol{S}_b$ 和 $\boldsymbol{S}_w$ 的广义瑞利商 (generalized Rayleigh quotient)．这种最优化问题存在解析解，即为

$$\boldsymbol{w} \propto \boldsymbol{S}_w^{-1}(\boldsymbol{\mu}_0 - \boldsymbol{\mu}_1)$$

这就是双类别 LDA 所得到的降维方向．

3.1.2 多类别 LDA

有了双类别 LDA 的基础，下面我们可以把它推广为任意 C 个类别数据的 LDA．在这种情况下，我们希望找到 k 个维度 $\boldsymbol{w}_1, \cdots, \boldsymbol{w}_k$，使得不同类别的数据向这个 k 维子空间的投影分散得最开．这里的 $k \leq C-1$，这可以理解为 C 个类别中心点只被包含在了 $C-1$ 维空间中．

下面我们需要对双类别 LDA 里的几个概念进行推广．我们仍然可以定义类间散度矩阵，但和上一节的定义有一些区别．实际上，这里的定义才是真正的类间散度矩阵

注意到我们将这些矩阵叫做散度而非协方差的原因是它没有除以样本量．

$$\boldsymbol{S}_b = \sum_{i=1}^{C} n_i(\boldsymbol{\mu}_i - \boldsymbol{\mu})(\boldsymbol{\mu}_i - \boldsymbol{\mu})^\top$$

其中 n_i 为第 i 类的样本数，$\boldsymbol{\mu}$ 为整个数据的均值点．这实际上就是所有类别的均值点的“加权散度矩阵”． 而类内散度矩阵可以定义为

我们上一节强调两个类别样本量相同就是因为上一节我们没有用样本量加权．

$$\boldsymbol{S}_w = \sum_{i=1}^{C} \boldsymbol{S}_i = \sum_{i=1}^{C} \sum_{\boldsymbol{x} \in \mathcal{X}_i} (\boldsymbol{x} - \boldsymbol{\mu}_i)(\boldsymbol{x} - \boldsymbol{\mu}_i)^\top$$

也就是说 $\boldsymbol{S}_w$ 就是所有类内散度的和，它也相当于所有类内协方差的加权和．这样定义的原因是这两个散度矩阵分解了数据的全局散度 $\boldsymbol{S}$，即

$$\boldsymbol{S}_b + \boldsymbol{S}_w = \boldsymbol{S} = \sum_{i=1}^{n} (\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^\top$$

有了上面的对于类间和类内散度的推广，我们就可以定义出多类别 LDA 的目标函数．注意到这里我们有了多个维度，所以分子和分母应该定义为所有维度上投影的散度之和，即

$$J(\boldsymbol{w}_1, \cdots, \boldsymbol{w}_k) = \frac{\sum_{i=1}^{k} \boldsymbol{w}_i^\top \boldsymbol{S}_b \boldsymbol{w}_i}{\sum_{i=1}^{k} \boldsymbol{w}_i^\top \boldsymbol{S}_w \boldsymbol{w}_i}$$

如果我们把所有的 $\boldsymbol{w}$ 作为列向量，横向排列成一个矩阵 $\boldsymbol{W} = [\boldsymbol{w}_1, \cdots, \boldsymbol{w}_k]$，那么损失函数可以被写做

$$J(\boldsymbol{W}) = \frac{\mathrm{tr}(\boldsymbol{W}^\top \boldsymbol{S}_b \boldsymbol{W})}{\mathrm{tr}(\boldsymbol{W}^\top \boldsymbol{S}_w \boldsymbol{W})}$$

$\mathrm{tr}\boldsymbol{A}$ 称为矩阵 $\boldsymbol{A}$ 的 **迹** (trace)，表示 $\boldsymbol{A}$ 的所有主对角线元素的和，即

$$\mathrm{tr}\boldsymbol{A} = \sum_i a_{ii}$$

这个最优化问题也有解析解，即 $\boldsymbol{w}_1, \cdots, \boldsymbol{w}_k$ 为 $\boldsymbol{S}_w^{-1}\boldsymbol{S}_b$ 的前 k 个特征向量（依照特征值降序排列）．需要指出的是，这里的 $\boldsymbol{w}$ 之间不一定是正交的，在这点上 LDA 与 PCA 和因子分析非常不同．

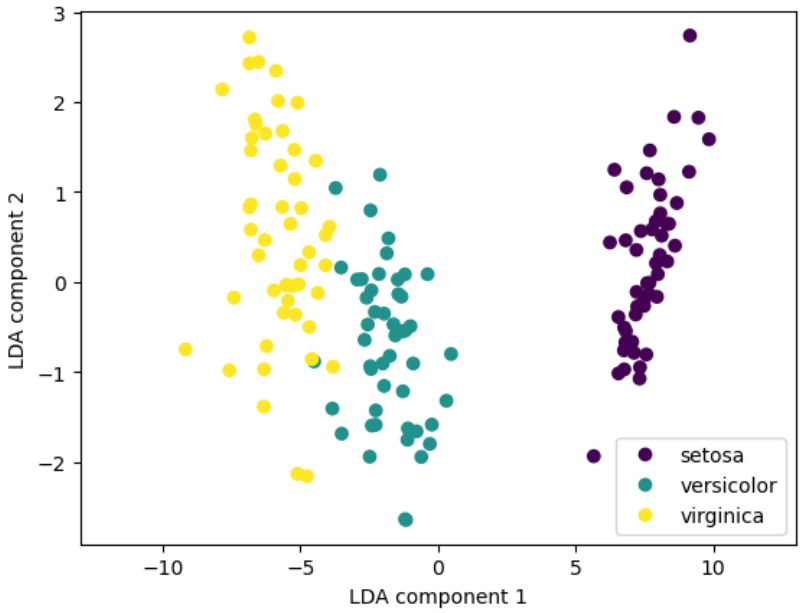
实际上这个矩阵至多有 $C-1$ 个非零特征值，这也是降维后不能超过 $C-1$ 维的原因．

3.1.3　LDA 的代码实现

LDA 可以用和前面的 PCA 和因子分析相同的语法来实现，只不过需要同时传入特征和标签两个变量．

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda = LinearDiscriminantAnalysis()
X_lda = lda.fit_transform(X, y)
```

其结果如下图所示．这三个类别之间似乎没有比使用 PCA 更加分离，这是因为 Iris 数据集的三个类别分开的方向本身就是数据方差较大的方向，因此二者结果差不多．但在如果类别之间分离的方向和方差最大的方向不同，例如 3.1.1 节的图中，PCA 与 LDA 的结果就会有较大差异．



4 无监督非线性降维方法

前面我们介绍了几种常见的线性降维算法．线性降维虽然具有简单、可解释性强等优势，但有时数据并不分布在一个线性子空间中．此时，我们需要使用非线性降维方法．

4.1 t-SNE 与 UMAP

本节我们来讨论科研数据可视化中最常见的两种降维方法—— t-SNE 和 UMAP．

4.1.1 t-SNE

我们在这里介绍的第一个算法称为 t-随机近邻嵌入 (t-stochastic neighbor embedding, **t-SNE**)．与前面介绍的几种算法不同，t-SNE（还有后面的 UMAP）并不试图找到降维的函数，或者说并不试图找到数据点所存在的子空间，而是直接得到每个数据点降维后的低维坐标 $\boldsymbol{z}_i$．其目标是使得数据点 $\boldsymbol{x}_i$ 之间的近邻关系在降维后得到保持——也就是说，邻近的数据点仍然保持邻近，而较远的数据点仍保持较远．

下面我们来看一看 t-SNE 的具体统计学原理．t-SNE 使用概率分布来描述数据之间的近邻关系．对于原始数据 $\boldsymbol{x}_i$，我们定义二维条件概率分布 $p_{i|j}$ 为数据点之间距离的归一化的高斯核函数，它描述数据点 $\boldsymbol{x}_i$ “有多大程度视 $\boldsymbol{x}_j$ 为邻居”

$$p_{i|j} = \frac{\exp(-||\boldsymbol{x}_i - \boldsymbol{x}_j||^2/2\sigma_i^2)}{\sum_{k\neq i}\exp(-||\boldsymbol{x}_k - \boldsymbol{x}_i||^2/2\sigma_i^2)}$$

这里使用高斯函数使得邻近的点的得分更高，而相距较远的点得分接近零．而参数 σ_i 用于调整邻居距离的“阈值”． σ_i 的选择我们后面马上会进行讨论．

这里的条件概率是不对称的，即 $p_{i|j} \neq p_{j|i}$． 为了使算法更简单，降低后面优化的复杂度，t-SNE 将分布对称化，得到二维概率分布 p_{ij}

选择这种对称方式而非直接求联合概率分布是为了使得离群值也被合理地考虑．

$$p_{ij} = \frac{1}{2n}(p_{i|j} + p_{j|i})$$

同样地，对于降维后的坐标 $\boldsymbol{z}_i$，我们也可以定义概率分布 q_{ij} 来描述数据点 $\boldsymbol{z}_i$ 和 $\boldsymbol{z}_j$ 之间的近邻关系．这里最直接的想法是使用与 p_{ij} 相同的方式定义 q_{ij}． 然而，这样的降维效果并不好．t-SNE 的解决办法是使用与高斯分布类似的 t 分布来定义 q_{ij}．这也是 t-SNE 名字中字母 t 的来源．

这里使用高斯核会产生“拥挤问题”，也就是高维下邻近的数据点在低维下难以保持邻近．

$$q_{ij} = \frac{(1 + ||\boldsymbol{z}_i - \boldsymbol{z}_j||^2)^{-1}}{\sum_{k\neq l}(1 + ||\boldsymbol{z}_k - \boldsymbol{z}_l||^2)^{-1}}$$

有了描述邻近关系的两个概率分布，我们只需找到一组 $\boldsymbol{z}_i$，使得 q_{ij} 尽量与 p_{ij} 接近，这时 $\boldsymbol{z}_i$ 就是一个保持数据点邻近关系的最好的低维表示了．我们定义 t-SNE 的损失函数为 P 和 Q 这两个分布的 KL 散度．

Kullback-Leibler (KL) 散度是对两个概率分布之间的“距离” 的描述，不过它不是对称的．

$$\mathcal{L} = D_{\mathrm{KL}}(P||Q) = \sum_{i,j} p_{ij}\log\frac{p_{ij}}{q_{ij}}$$

接下来，我们可以通过梯度下降的方式求解这个最优化问题．这就是 t-SNE 降维的完整过程．

我们讲完了 t-SNE 的推导过程，但其中有一个自由参数还没有解释，那就是高斯分布 $p_{i|j}$ 里的 σ_i．我们说过，它代表每个点“邻居距离的阈值”．而在使用 t-SNE 时，我们并

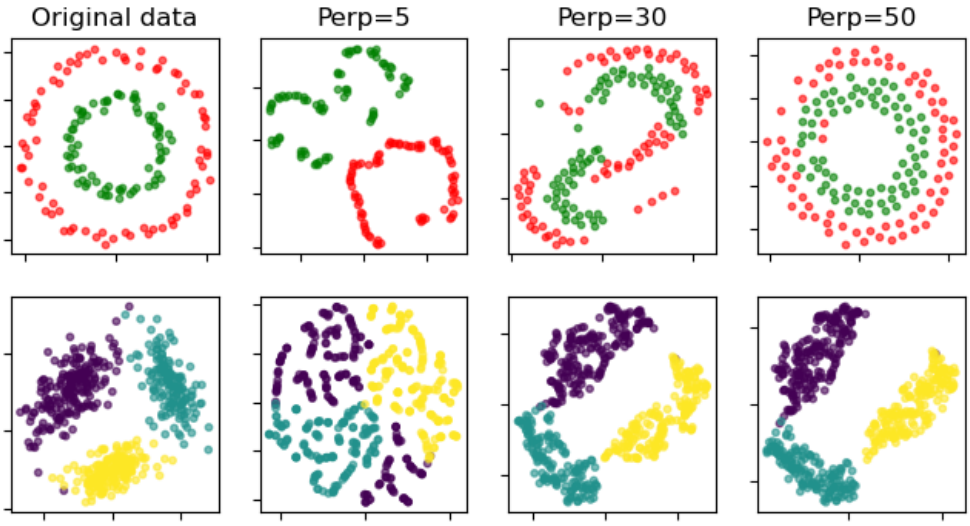
不直接指定各个 σ_i 的值，而是指定一个称为 **困惑度** (perplexity) 的参数，它代表的是各个点的“邻居数”．在给定了一个困惑度后，它以如下的方式决定每个点的 σ_i．

这里 $-\sum_j p_{j|i}\log_2 p_{j|i}$ 就是点 i 的近邻分布的信息熵 $H(P_i)$.

$$\mathrm{Perp} = 2^{-\sum_j p_{j|i}\log_2 p_{j|i}}$$

困惑度一般选择在 5-50 之间，根据降维后可视化效果选择． 下图显示了两个数据集使用不同困惑度的降维结果．可以发现，困惑度较低时，降维后的数据点分布较为均匀，而随着困惑度的升高，降维后逐渐产生聚类．我们一般选择一个较为折衷的结果．另外，从上半图可以发现，较大的困惑度可以捕捉到数据的整体结构，而较小的困惑度则会只关心局部结构．

这个网站 以交互的方式呈现了一些不同的数据集下各个困惑度的降维结果.



4.1.2　UMAP

UMAP (uniform manifold approximation and projection) 是 t-SNE 的一个改良版本，由于其性能整体更优，现在几乎已经全面取代了后者． UMAP 的原始论文是以范畴论和图论的语言描述的，较为艰深晦涩．然而，UMAP 的思路本质上和 t-SNE 是几乎完全一样的，主要是做了一些技术上的改进．下面我们就来简单看一下 UMAP 做了哪些改进．

UMAP 的思路也是对高维和低维数据进行表示，并让这两个表示尽量相似．但与 t-SNE 不同的是，在 UMAP 中，每个数据点都有一个明确的“邻居” 划分，并且它们只关心自己的邻居． 这里我们需要设定一个超参数 $n_{\text{neighbors}}$，离每个数据点最近的这么多个其他数据点会成为它的邻居．下面，我们仍然使用一个函数 $w_{j|i}$ 来描述数据点之间的近邻关系，而非邻居之间的近邻关系被直接设为 0．在此处，UMAP 不要求这个函数是一个概率分布，也就是说不对它进行归一化．

$n_{\text{neighbors}}$ 在一些地方也会被记为 k.

$$w_{j|i} = \begin{cases} \exp\left(\frac{-\max(0,||\boldsymbol{x}_i-\boldsymbol{x}_j||-\rho_i)}{\sigma_i}\right), & \boldsymbol{x}_i \text{ 和 } \boldsymbol{x}_j \text{ 是邻居} \\ 0, & \text{otherwise} \end{cases}$$

其中 ρ_i 和 σ_i 是超参数， 但是我们并不直接设定这两个超参数，而是算法会根据我们设定的 $n_{\text{neighbors}}$ 自动选择．

具体设定方式详见 UMAP 原始论文.

下面，UMAP 也对这个函数进行对称化，不过对称化的方式也有些不同．这种对称化的方式来自图论．

$$w_{ij} = w_{i|j} + w_{j|i} - w_{i|j}w_{j|i}$$

而对于降维后的 $\boldsymbol{z}_i$ 的近邻关系的表示（称为“模糊” 表示），UMAP 不再使用 t 分布，而使用一种“平顶” 的指数衰减函数 v_{ij}. 这里我们需要设定另一个超参数 $\min_{\text{dist}}$，当两个点的距离小于 $\min_{\text{dist}}$ 时，它们的近邻关系不再上升. 不过这个函数是不可微的，因此实际使用的是它的一个近似函数.

$$v_{ij} = \begin{cases} 1, & ||\boldsymbol{z}_i - \boldsymbol{z}_j|| \leq \min_{\text{dist}} \\ \exp(-||\boldsymbol{z}_i - \boldsymbol{z}_j|| + \min_{\text{dist}}), & \text{otherwise} \end{cases}$$

最后，UMAP 使用两个近邻表示之间的二元交叉熵而非 KL 散度作为损失函数

$$\mathcal{L} = \text{BCE}(W, V) = \sum_{i,j} \left(w_{ij} \log \frac{w_{ij}}{v_{ij}} + (1 - w_{ij}) \log \frac{1 - w_{ij}}{1 - v_{ij}} \right)$$

UMAP 的超参数相对容易理解一些—— $n_{\text{neighbors}}$ 与 t-SNE 的困惑度类似，它用于平衡数据的局部结构与全局结构. 而 $\min_{\text{dist}}$ 则表示降维后数据点的离散程度.

UMAP 超参数的交互性演示见 这个网站.

4.1.3 t-SNE 和 UMAP 的几点说明

t-SNE 和 UMAP 与我们看到的几种算法有非常大的不同，因此我们在此对于它们的使用需要进行几点说明.

首先，在 t-SNE 和 UMAP 的降维结果中，数据点之间的距离是没有意义的. 因此，我们不能比较两个聚类的大小，或者比较聚类之间的距离等. 我们唯一可以比较的是两个数据点是否属于同一个聚类，同一个聚类的数据点往往在原始高维空间中相聚更近.

我们在讲解 t-SNE 时说过，降维的结果往往会自动产生聚类 (clusters)，UMAP 也是如此.

第二，t-SNE 和 UMAP 对超参数的敏感性较高，因此使用时应尝试多个超参数组合，选择最好的结果. 由于 t-SNE 和 UMAP 最常见的目的就是数据可视化，因此一般人工选择一种局部聚类和全局结构平衡较好的结果. 同时，这两种算法的优化过程中都有一定的随机性，因此即便是用了同一组超参数，每次降维的结果也可能是不一样的. 为了保证结果的稳定性，使用时应该多跑几次，选择较为稳定的结果.

第三，我们前面讲解降维原理时都使用了点之间的欧几里得距离 $||\boldsymbol{x}_i - \boldsymbol{x}_j||_2$. 但 t-SNE 和 UMAP 可以使用任何我们指定的距离度量. 不过我们一般只是在原始的高维空间中选择合适的距离度量，在降维后的低维空间中还是习惯使用欧几里得度量，这样能更好地可视化数据之间的相似性.

最后，t-SNE 和 UMAP 算法虽然思路十分类似，但其结果仍有一定的不同. 一般认为，UMAP 比 t-SNE 能更好地保留数据的全局结构. 同时，UMAP 的运行速度也更快：t-SNE 的时间复杂度为 $O(n^2)$，而 UMAP 的时间复杂度约为 $O(n^{1.14})$. 因此，UMAP 几乎已经全面取代了 t-SNE. 不过，UMAP 在少数结构比较特殊的数据集上的结果不如 t-SNE.

UMAP 不如 t-SNE 的例子仍然见上面的网站.

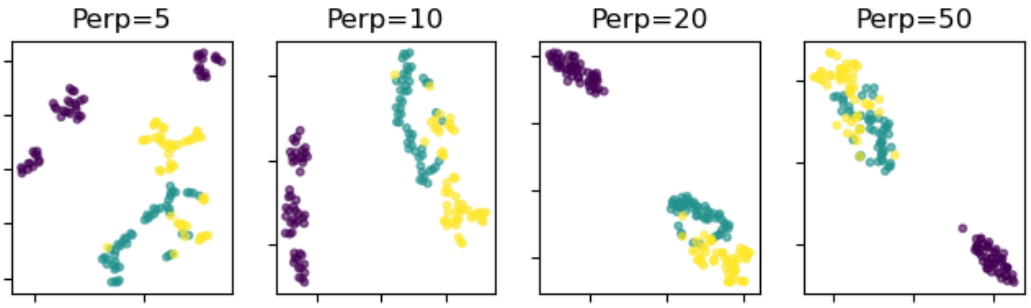
以上是 t-SNE 和 UMAP 的几点异同. 那么与后面我们将要介绍的其他非线性降维算法相比，该在什么情况下使用 t-SNE 和 UMAP 呢？我们在前面已经看到，这两种算法在意的是数据之间的近邻结构，而不是数据所在的低维子空间——即流形（即使 UMAP 的名字里有流形这个词）. 这也体现了我们该在何时使用它们. 例如，在生物学中，t-SNE 和 UMAP 是对单细胞测序（scRNAseq）等组学数据可视化的标准降维算法. 这是因为 scRNAseq 的结果体现了细胞类型，相同类型的细胞会形成聚类. 而我们可视化想要体现的往往就是这些不同的细胞类型. 此时 t-SNE 和 UMAP 就是合适的降维算法.

4.1.4　t-SNE 和 UMAP 的代码实现

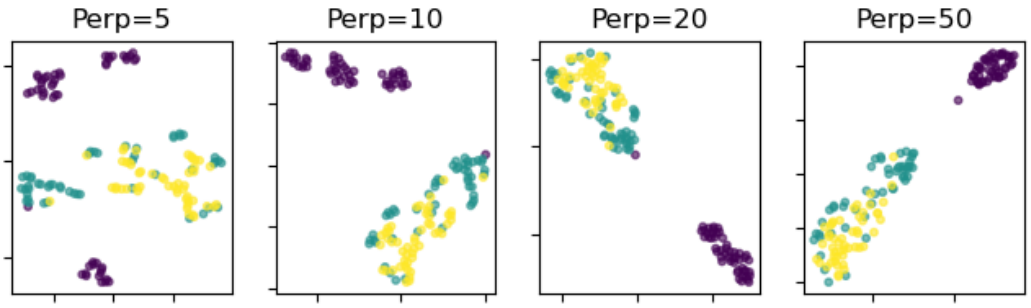
t-SNE 算法在 sklearn 里实现为 `sklearn.manifold.TSNE`，其语法还是与前面的几种算法类似．注意 t-SNE 和 UMAP 不需要对数据中心化，但我们仍然需要使用标准化或归一化的数据，这是因为我们不希望某一个或几个数值特别大的变量主导了距离度量，如 1.1.2. 节所述．

```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, perplexity=10, n_iter=1000)
X_tsne = tsne.fit_transform(X_standardized)
```

我们也可以多尝试几种困惑度的值，一起画出来比较．其结果如下图所示．可以发现，越大的困惑度意味着聚类越明显．由于这个数据没有明显的全局结构，因此在这方面的区别不大．在下图中，困惑度为 10 时较好地平衡了数据点的聚类．



我们说过，t-SNE 可以使用欧几里得度量以外的其他度量，这可以通过超参数 `metric` 进行调节．例如，我们可以使用余弦相似度作为距离度量，这时只需在实例化算法时指明 `metric="cosine"` 即可．这里使用余弦相似度时，不同的类别不能较好地分开，这是因为在这个数据集中余弦相似度并不是一个合适的度量．不过我们这里只是使用鸢尾花数据集举个例子而已．



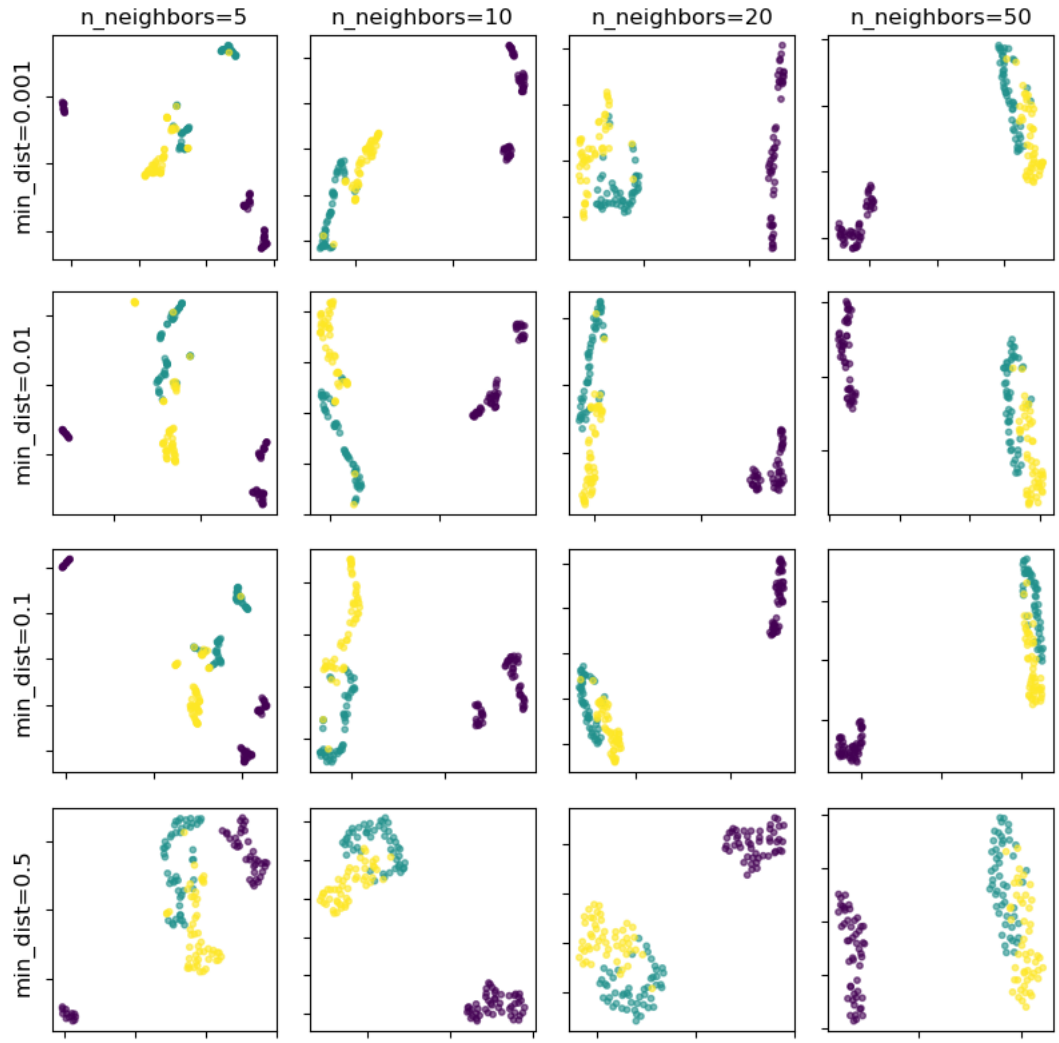
还需要指出的一点是，sklearn 的 `TSNE` 算法只有 `fit_transform()` 方法，而没有独立的 `transform()` 方法．也就是说，我们无法在训练之后把新的数据点按照刚才训练出的降维方式进行降维．这是因为 t-SNE 并不会得到降维的函数，而是直接得到了训练数据降维后的坐标．因此，它无法直接处理新的数据点．不过，实际上是有办法估计新数据点的低维坐标的，只不过这个方法在 sklearn 中并没有实现．其他的一些 t-SNE 库和我们马上将看到的 UMAP 库则实现了这个方法．

UMAP 算法较新，因此在 sklearn 里没有实现．这里我们使用 UMAP 原作者编写的 `umap` 库． 它的语法也与 sklearn 里的算法类似．

UMAP 库的 github 网页见 这里.

```python
from umap import UMAP

umap = UMAP(
    n_components=2,
    n_neighbors=10,
    min_dist=0.3
  )
X_umap = umap.fit_transform(X_standardized)
```

这里重要的超参数包括 `n_neighbors` 和 `min_dist`. 我们前面说过，$n_{\text{neighbors}}$ 与 t-SNE 里的困惑度类似，调节降维关注的是局部结构还是全局结构，一般选择在 5-50 之间. 而 min_{dist} 则决定了降维后数据点之间的离散程度，一般选择在 0.001-0.5 之间. 下图显示了两个超参数的不同组合下的降维结果.



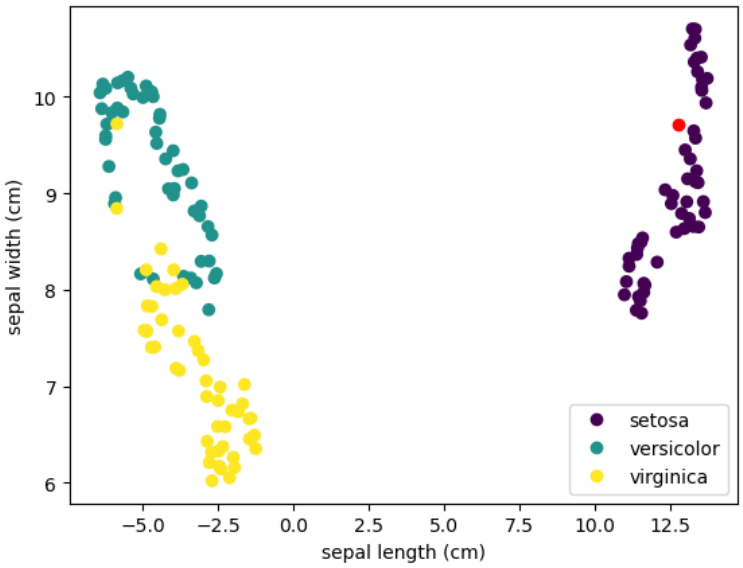
可以看到，在这个鸢尾花数据集中，大约在 `n_neighbors=20, min_dist=0.1` 附近的降维效果最好. 此时各个类别没有被撕裂，同时数据点的聚集程度又适中.

UMAP 同样也可以使用其他的度量，我们在这里就不再举例了. 同时，UMAP 还实现了 `transform()` 方法，可以用于对新的数据点的降维. 由于 UMAP 并不会得到降维的函数，因此这个方法实际上是间接地对新数据点的低维坐标的估计，其具体过程我们在这里不再赘述.

```
umap = UMAP(
    n_components=2,
    n_neighbors=20,
    min_dist=0.1
  )
X_umap = umap.fit_transform(X_standardized)
x_new_umap = umap.transform(scaler.transform(x_new))
```

上面的代码对一个新的数据点进行了降维，从下图可以看到它成功地被归为了 *I. setosa* 类别中．

同一个数据点的 PCA 降维见 2.1.2. 节．



4.2　MDS 与 Isomap

我们说过，非线性数据降维又称为流形学习，这是因为降维就是在寻找数据所在的低维流形．不过，前面的 t-SNE 和 UMAP 算法的思路都只是在保持邻近关系，而并不关心数据所在的流形．本节我们来介绍一个直接“展开并压平” 数据所在的流形的算法，即 Isomap．它也是非常常用的非线性降维算法之一．

4.2.1　多维尺度变换

在介绍 Isomap 之前，我们先来看一个更为简单而基础的降维算法，称为多维尺度变换 (multidimensional scaling, **MDS**)．它是 Isomap 降维的基础之一．

MDS 的核心目标是保持数据点之间的距离． 也就是说，MDS 试图使得降维后数据点两两之间的距离 d'_{ij} 尽量接近原始高维数据点间的距离 d_{ij}．那么我们可以把 MDS 的损失函数写作

在数学上，保持距离不变的映射称为等距映射 (isometric map)．

$$\mathcal{L} = \sum_{i,j} \left(d'_{ij} - d_{ij}\right)^2$$

我们可以通过我们熟悉的梯度下降法来求解．这里的度量还是可以取任意一种度量．不过，当在低维空间中使用欧几里得度量 $d'_{ij} = ||\boldsymbol{z}_i - \boldsymbol{z}_j||_2$ 时，这个问题有闭式解．下面我们就来推导一下．

下面我们仍把数据降维后的低维坐标记作 $\boldsymbol{z}_i$．那么在理想情况下，若数据之间的距离完全得以保持，也就是说 $d_{ij} = d'_{ij}$，那么有

$$d_{ij}^2 = ||\boldsymbol{z}_i - \boldsymbol{z}_j||^2 = ||\boldsymbol{z}_i||^2 + ||\boldsymbol{z}_j||^2 - 2\boldsymbol{z}_i\boldsymbol{z}_j^2$$

由于数据点可以任意整体平移而不改变其距离，我们可以让降维后的数据点是中心化的，也就是 $\sum_i \boldsymbol{z}_i = \boldsymbol{0}$．此时，将上式对 i 和 i, j 求和，可以得到

$$\sum_i d_{ij}^2 = \sum_i ||\boldsymbol{z}_i||^2 + n||\boldsymbol{z}_j||^2$$
$$\sum_{i,j} d_{ij}^2 = \sum_{i,j} ||\boldsymbol{z}_i||^2 + n\sum_j ||\boldsymbol{z}_j||^2 = 2n\sum_i ||\boldsymbol{z}_i||^2$$

这样我们可以将 $||\boldsymbol{z}_i||$ 代换为 d_{ij}^2 的和式．我们定义剩余的交叉项 $b_{ij} = \boldsymbol{z}_i\boldsymbol{z}_j^\top$，那么有 $b_{ij} = \frac{1}{2}(||\boldsymbol{z}_i||^2 + ||\boldsymbol{z}_j||^2 - d_{ij}^2)$．代换后得到

$$b_{ij} = \frac{1}{2}\left(\frac{1}{N}\sum_k \left(d_{kj}^2 + d_{ik}^2\right) - \frac{1}{N^2}\sum_{k,l} d_{kl}^2 - d_{ij}^2\right)$$

注意到 b_{ij} 的矩阵形式 $\boldsymbol{B} = \boldsymbol{Z}\boldsymbol{Z}^\top$，用它可以求解出低维坐标 $\boldsymbol{Z}$．由于 $\boldsymbol{B}$ 是一个对称矩阵，我们可以将其对角化为 $\boldsymbol{B} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^\top$．那么 $\boldsymbol{Z}$ 的一个解为

$\boldsymbol{\Lambda}$ 是一个对角元非负的对角矩阵，我们只需将每个对角元开方，即可得到开方的矩阵．

$$\boldsymbol{Z} = \boldsymbol{V}\boldsymbol{\Lambda}^{1/2}$$

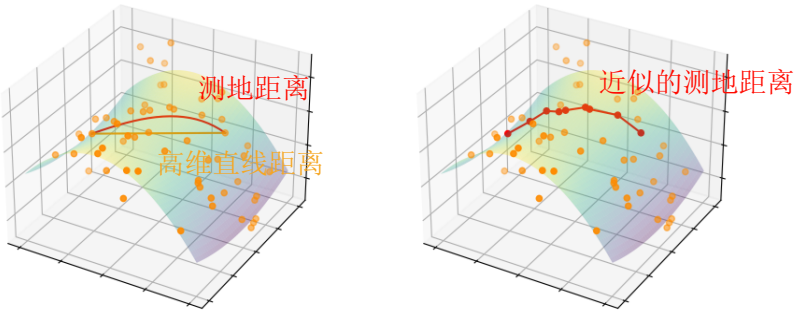
此时的 $\boldsymbol{Z}$ 是一个 $n \times n$ 的矩阵，每行是降维后的每个数据点的坐标．也就是说我们将维度降至 n 维时，我们可以保证降维是严格等距的．

最终得到的坐标可以任意进行旋转，也就是乘上任意一个正交矩阵．

若我们需要降至更低的 k 维，问题会复杂一些，并不存在闭式解．但近似来看，我们可以在对角化时将特征值降序排列，将 $\boldsymbol{Z}$ 只取前 k 列．这种解法称为经典 MDS 或 Torgerson's MDS．可以证明，经典 MDS 实际上与 PCA 等价，但它提供了一种不同的出发点——只需提供数据点之间两两的距离，就可以在低维尽量等距地重构这些数据点．

4.2.2　Isomap

等距特征映射 (isometric feature mapping, **Isomap**) 是一种非常经典的流形学习算法．Isomap 算法以 MDS 为基础，也试图寻找一个尽量等距的降维映射．但考虑到数据处于一个低维流形上，Isomap 并不使用高维的直线距离，而是使用两点在流形上的距离，称为**测地距离** (geodesic distance)．以测地距离作为输入，在低维仍用欧式距离进行重构，其结果就类似于“展平” 了数据所在的流形，这就是 Isomap 的基本思想．



与我们的直观理解类似，流形上的测地距离可以定义为连接两点在流形上的最短路径的长度．不过，我们并不知道数据所在的流形具体是长什么样子的，所以无法直接求得测地距离．不过，Isomap 假设数据点布满了整个流形，此时我们可以通过已有的数据点来近似出测地距离——我们可以用中间的一些数据点，在两点之间画出一条最短的折线．如上面右图所示．当然，我们直接连接的两个点不能离得太远，不然就不能保证连线近似位于流形之上了．这样，我们能画出的最短连线的长度就是近似的测地距离．

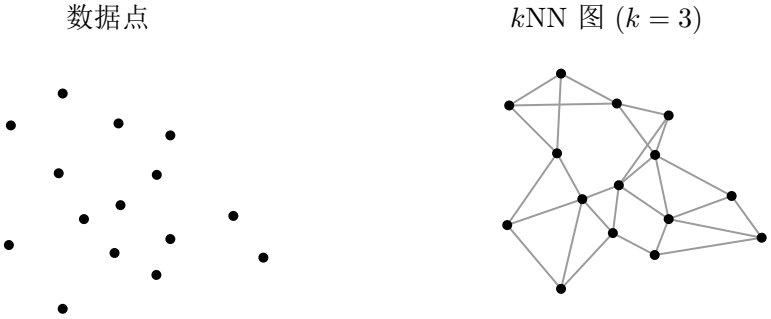
不严格地说，这种最短路径被称为测地线 (geodesic)．

从具体算法来说，Isomap 将邻近的数据点连接起来，建立一个 **无向图** (undirected graph)，称为近邻图．而所谓“邻近的点” 的选择标准有两种：

- ϵ-阈值图：若两个点之间的距离不超过 ϵ，则连接它们；
- k 近邻 (kNN) 图：每个点都与与自己最近的 k 个点连接．

kNN 图本身是一种有向图，但在 Isomap 中视为无向图．

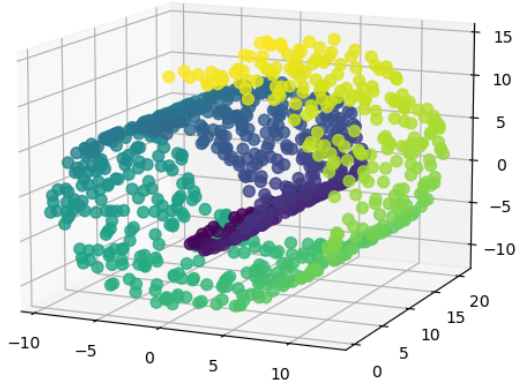
其中 kNN 图更加常用一些．下面显示了一个 kNN 图的例子．



有了近邻图后，我们就可以搜索两点之间的最短路径，用以近似测地距离．这已经有较为成熟的图论算法，如 Dijkstra 算法或 Floyd-Warshall 算法，在此不再赘述．计算出每一对数据点之间的近似测地距离后，将其输入进 MDS 算法中，得到低维嵌入，这就是 Isomap 的整个过程．

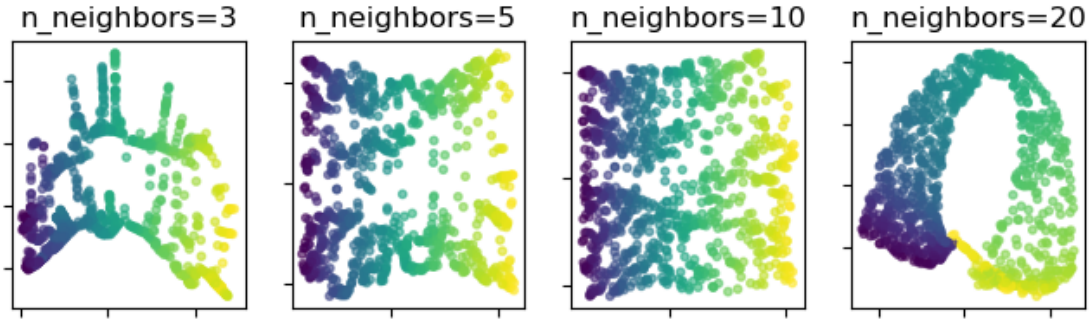
4.2.3　Isomap 的代码实现

为了更好地体现 Isomap 对流形结构的处理，我们在此不使用鸢尾花数据集，而是构造一个明显流形结构的数据． sklearn 提供一个函数 `sklearn.datasets.make_swiss_roll()`，可以构造一个“瑞士卷” 数据．例如 `X, y = datasets.make_swiss_roll(n_samples=1000)` 可以构造出如下数据．其中标签 y 用颜色来表示．



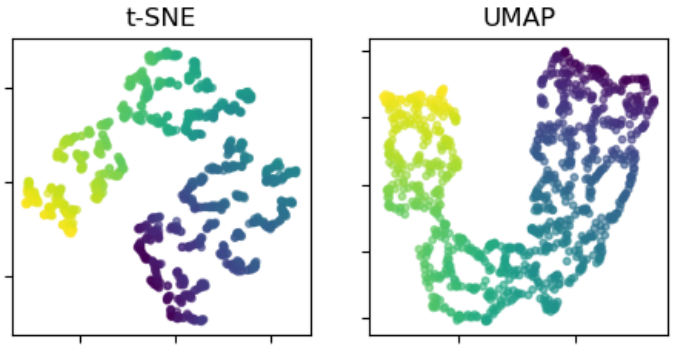
下面我们来使用 Isomap 算法对瑞士卷数据进行降维．sklearn 里的 Isomap 算法位于 `sklearn.manifold.Isomap`．它的重要超参数除了我们熟悉的 `n_components` 还有 `n_neighbors`，它代表建立 kNN 图时每个点的邻居数，即 k．当 `n_neighbors` 较小时，图的连接较为稀

疏，连接两点的最短折线会有很多重叠，因此降维结果会出现很多排列成线的情况．而当 `n_neighbors` 较大时，估计的测地距离会更接近原始的高维距离，因此降维结果会体现出数据在高维中的分布，而并没有“展开” 流形．下图显示了不同 `n_neighbors` 时的降维结果．可以看出，`n_neighbors` 位于 10 左右时的降维效果最好．



相比之下，前面讨论的 t-SNE 和 UMAP 由于在意的是数据的聚类而非流形结构，它们在这种数据集上的降维效果不如 Isomap． 下面显示了 t-SNE 和 UMAP 在较合适的超参数下的降维结果．通过这个例子，我们可以看出何时应该选择 t-SNE 或 UMAP，何时应该选择 Isomap．

图中 t-SNE 的超参数为 Perp = 10，UMAP 的超参数为 $n_{\text{neighbors}} = 8, \min_{\text{dist}} = 0.5$.



4.3　局部线性嵌入

本节介绍另一种“展平” 流形的降维方法，称为 **局部线性嵌入** (locally linear embedding, **LLE**).

4.3.1　局部线性嵌入

局部线性嵌入正如其名，其基本思想就是保持流形局部的线性关系——对于每一个点 $\boldsymbol{x}_i$，它和它的一些邻居 $\boldsymbol{x}_j$ 们应该有一个近似的线性关系 $\boldsymbol{x}_i \approx \sum_j w_{ij}\boldsymbol{x}_j$，其中 w_{ij} 是线性组合的系数．这个系数可以通过最小化线性组合误差的方式求解，即

$$\text{找到系数 } w_{ij} \text{ 以最小化 } \sum_i \left\| \boldsymbol{x}_i - \sum_{\text{邻居 } \boldsymbol{x}_j} w_{ij}\boldsymbol{x}_j \right\|^2$$

这里的邻居一般也是通过 k 近邻的方式定义，即人工指定邻居数 k，算法自动找到与每个点最接近的 k 个数据点作为邻居．而对于系数 w_{ij} 我们一般会给一个约束条件 $\sum_j w_{ij} = 1$．这样，每个点的坐标都可以视为其邻居坐标的近似加权平均，w_{ij} 即为 $\boldsymbol{x}_j$ 的权重．

而下一步，就是找到低维坐标 $\boldsymbol{z}_i$，使得上面的近似线性关系被尽量保持．也就是说，在给定了上面求出的系数 w_{ij} 后，进而

注意此处的邻居 $\boldsymbol{z}_j$ 是上面高维找到的邻居的低维坐标，而非在低维重新找一次邻居．

$$\text{找到低维坐标 } \boldsymbol{z}_i \text{ 以最小化 } \sum_i \left\| \boldsymbol{z}_i - \sum_{\text{邻居 } \boldsymbol{z}_j} w_{ij} \boldsymbol{z}_j \right\|^2$$

这样找到的低维坐标 $\boldsymbol{z}_i$ 即为 LLE 的降维结果．不过注意到这里的低维坐标仍然有平移不变性，因此我们一般也需要求中心化 $\sum_i \boldsymbol{z}_i = \boldsymbol{0}$.

与前面的很多降维算法类似，LLE 是有闭式解的．不过其所需的数学知识稍高，我们在这就不进行推导了，而是直接给出结论．我们定义矩阵

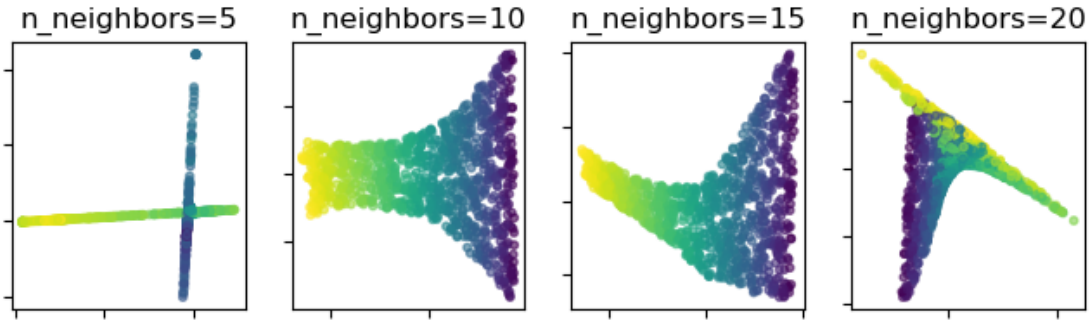
$$\boldsymbol{M} = (\boldsymbol{I} - \boldsymbol{W})^\top (\boldsymbol{I} - \boldsymbol{W})$$

那么 $\boldsymbol{M}$ 的最小的 k 个特征值所对应的特征向量所组成的矩阵即为降维后的坐标矩阵 $\boldsymbol{Z}$.

LLE 关注的是数据的局部结构，而不像 Isomap 一样关注整体结构，因此它的时间复杂度也更低一些．Isomap 的时间复杂度为 $O(n^3)$，而 LLE 只有 $O(n^2)$.

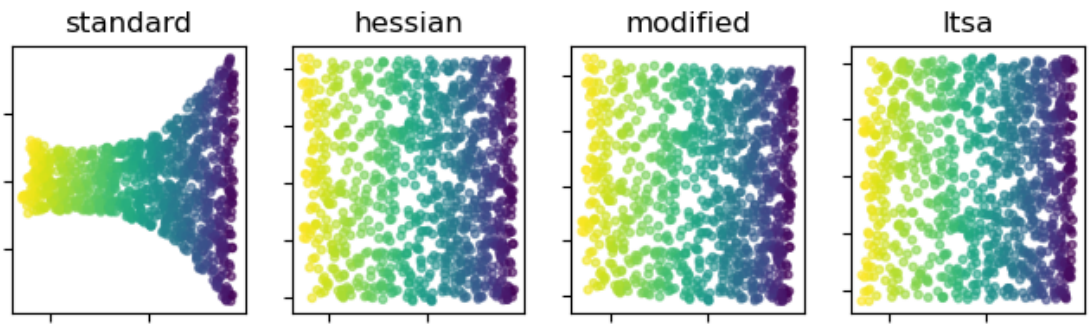
4.3.2　LLE 的代码实现

我们在此仍然使用上面的瑞士卷数据来演示 LLE 的降维．LLE 在 sklearn 里位于 `sklearn.manifold.locallyLinearEmbedding`．其重要超参数也包括 `n_components` 和 `n_neighbors`．下图显示了不同 `n_neighbors` 下的降维结果．可以看到，LLE 降维结果受邻居数的影响与 Isomap 类似．



LLE 还有很多的变种，主要包括 Hessian LLE (HLLE)、modified LLE (MLLE) 和局部切空间排列 (Local Tangent Space Alignment, LTSA)．在 sklearn 中，我们可以使用超参数 `method` 来调用不同的 LLE 变种，而默认的则是最原始的 LLE，即 `method='standard'`．从下图可以看到，几种 LLE 变种的降维效果都优于原始的 LLE．它们的具体原理我们就不细讨论了．

这里都使用了各自最优的邻居数．

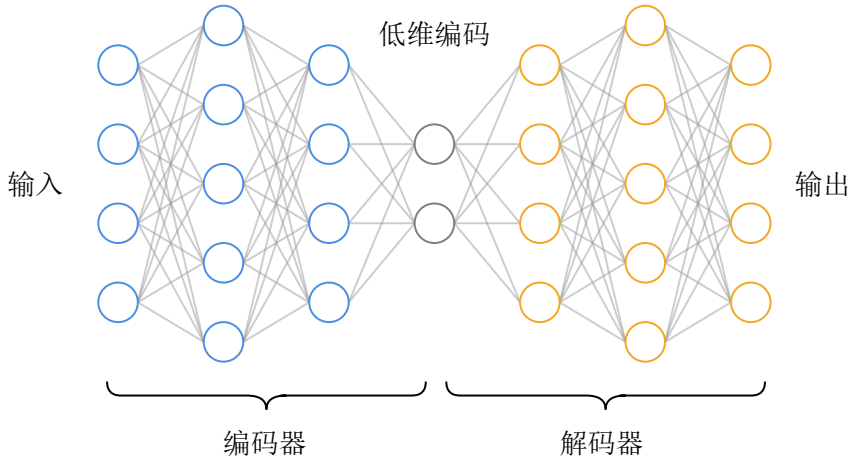


4.4　自编码器

最后，我们来介绍一种统计思维含量最低、但是又最复杂、最强大的非线性降维算法——**自编码器** (autoencoder).

4.4.1　自编码器的结构

自编码器与前面的传统统计学降维算法不同，它实际上是以深度学习的思路进行降维．它使用一种特殊的神经网络，其基本结构如下图所示．



自编码器是一个前馈神经网络，它的输入层和输出层的宽度等于数据的原始维数，而它的中间有一个“瓶颈”，瓶颈的宽度就是我们想降到的维数．瓶颈的左侧称为 **编码器**，右侧称为 **解码器**．数据点的坐标从输入层输入，经过编码器计算到中间的瓶颈层，此层的神经元激活值即为该数据点的低维坐标．这个过程称为编码或“压缩”．而右侧的解码器则可以从低维坐标逆向“重构” 回高维坐标．这个过程称为解码或“解压”．

与其他算法相同，自编码器也需要根据数据进行训练．自编码器训练的目标是：让输出与输入尽量接近，即损失函数为

这种特殊的无监督学习可以称为**自监督学习**.

$$\mathcal{L} = \sum_i ||\boldsymbol{x}_i - \mathrm{AE}(\boldsymbol{x}_i)||^2$$

这里 $\mathrm{AE}(\boldsymbol{x}_i)$ 就是整个自编码器最右侧的输出．这样，如果我们只看右侧的解码器的话，解码器会试图使用瓶颈处的低维信息尽量重构出输入．这就使得自编码器会在瓶颈处保留最重要的信息．因此，自编码器学习到的是一种对高维信息的有效低维编码，或者说一种隐藏变量．这个思路与 t-SNE/UMAP 和 Isomap/LLE 都不同．

4.4.2　一个简单的自编码器的实现

由于自编码器是一种神经网络，它在 sklearn 中没有实现．我们需要使用深度学习框架来进行实现．此处我们选择目前最常用的深度学习框架 PyTorch．同时，为了体现自编码器的强大之处，我们在此使用一个更为复杂的数据集，即著名的 MNIST 手写识别数据集．MNIST 数据集包含了一系列 28×28 像素的灰度手写数字图片．如下图所示．每个图片相当于 $28 \times 28 = 784$ 维空间中的一个点．我们下面用自编码器把它降至 2 维．

我们通过下面的代码从 PyTorch 中导入 MNIST 数据集．我们这里导入了训练集和测试集，并自动导入为可以用于训练的 DataLoader 类型．

```
from torchvision.datasets import MNIST
from torchvision import transforms

# Download the MNIST Dataset and put in data loader
trainset = MNIST(root="./data", train=True, download=True,
                transform=transforms.ToTensor())
trainloader = torch.utils.data.DataLoader(dataset=trainset,
                batch_size=1024, shuffle=True)

testset = MNIST(root="./data", train=False, download=True,
                transform=transforms.ToTensor())
testloader = torch.utils.data.DataLoader(dataset=testset,
                batch_size=32, shuffle=False)
```

下面我们来构造我们的自编码器神经网络．我们此处使用最简单的神经网络，它由一系列全连接层构成，其中编码器每层的神经元数目分别为 784 - 128 - 32 - 8 - 2，并使用 ReLU 激活函数．解码器的结构与之完全对称．

编码器和解码器的结构通常是对称的，但这并不是必需的．

```
import torch
from torch import nn

class AutoEncoder(nn.Module):

  def __init__(self):
      super().__init__()

      # Encoder: 784 - 128 - 32 - 8 - 2
      self.encoder = nn.Sequential(
          nn.Flatten(),
          nn.Linear(784, 128),
          nn.ReLU(True),
          nn.Linear(128, 32),
          nn.ReLU(True),
          nn.Linear(32, 8),
          nn.ReLU(True),
          nn.Linear(8, 2),
          nn.ReLU(True)
      )

      # Decoder: 2 - 8 - 32 - 128 - 784
      self.decoder = nn.Sequential(
          nn.Linear(2, 8),
          nn.ReLU(True),
          nn.Linear(8, 32),
          nn.ReLU(True),
          nn.Linear(32, 128),
          nn.ReLU(True),
          nn.Linear(128, 784),
```

```
            nn.ReLU(True),
            nn.Unflatten(dim=1, unflattened_size=(1, 28, 28))
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

由于神经网络在 GPU 上的训练速度远高于 CPU，因此我们使用下面这行代码，在有 GPU 的 CUDA 框架时用 GPU 训练.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

下面的几个步骤是训练神经网络的固定步骤. 首先我们要实例化一个网络，并且选择损失函数和优化器. 这里我们选择原图和重构图的 **均方误差** (MSE) 作为损失函数，并选择常用的 Adam 优化器.

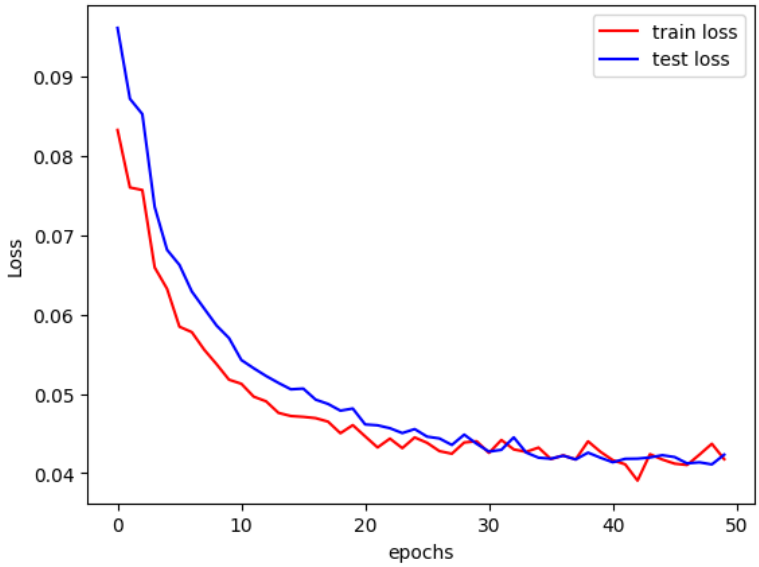
```
# Model Initialization
model = AutoEncoder().to(device)

# Validation using MSE Loss function
loss_function = nn.MSELoss()

# Using an Adam Optimizer
optimizer = torch.optim.Adam(model.parameters(),
                lr = 1e-3, weight_decay = 1e-10)
```

接下来就是对神经网络进行训练. 与其他神经网络相同，自编码器也需要注意过拟合的问题，因此我们不仅需要注意训练集上的误差，还需要关注测试集上的误差. 这里我们总共训练 50 个 epoch，在每个 epoch 后计算一下两个数据集的误差并记录下来.

```
epochs = 50
train_losses = []
test_losses = []

for epoch in range(epochs):

    model.train()
    for image, _ in trainloader:

        # Move the input tensor to device (if GPU)
        image = image.to(device)

        # Output of Autoencoder and calculate loss
        reconstructed = model(image)
        loss = loss_function(reconstructed, image)

        # Standard way to do backprop and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

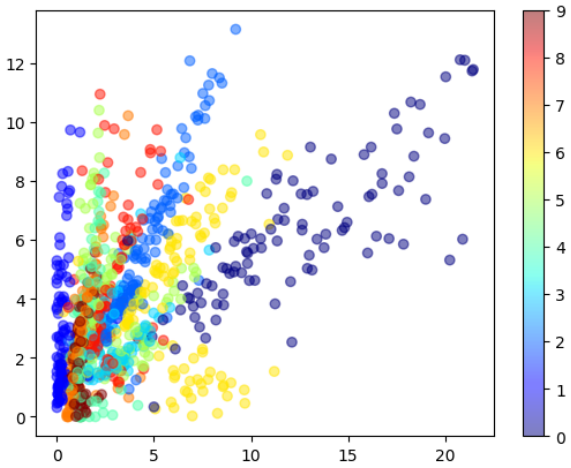
```
21
22      # Store the training loss
23      train_losses.append(loss.item())
24
25      # Evaluation mode, grad will not be calculated
26      model.eval()
27      with torch.no_grad():
28          for image, _ in testloader:
29
30              # Calculate loss, same as training
31              image = image.to(device)
32              reconstructed = model(image)
33              loss = loss_function(reconstructed, image)
34
35          # Store the test loss
36          test_losses.append(loss.item())
37
38      print('epoch = ' + str(epoch))
39
40  print('Training done!')
```

训练完成后，我们可以画出训练过程中训练集和测试集的误差变化．可以看到训练集的误差逐渐而下降，并在最后趋于稳定．这说明训练是成功的．同时，测试集的误差与训练集接近，最后并没有上升，这表示没有过拟合．

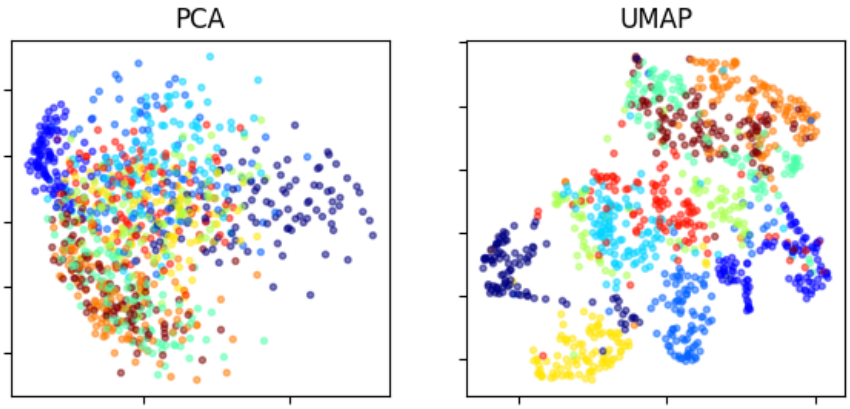


如果我们对比一下训练后神经网络的输入和输出，也可以发现神经网络较好地重构了输入的图片．下图中上面一行显示了原始图片，而下面一行显示了自编码器重构的图片．

在成功训练了我们的自编码器后，我们可以通过 `model.encoder()` 来调用编码器部分，以实现数据降维．下图显示了对部分图片的降维结果．可以看到，自编码器较好地分开了不同数字的图片．



最后，我们附上两个其他的经典降维算法——PCA 和 UMAP 对于同样的部分数据的降维结果．显然，PCA 作为线性降维算法，无法成功地区分不同的数字． UMAP 由于在意的是数据的聚类结构，因此还算较为成功地区分了不同的数字．不过它和自编码器在降维时在意的结构与自编码器不同，因此二者还是有不同的使用条件．



本笔记的参考包括：

1. scikit-learn 官方文档 (https://scikit-learn.org/stable/)
2. Stack Exchange 论坛，包括 Stats 1576
3. 周志华《机器学习》
4. UMAP 官方文档 (https://umap-learn.readthedocs.io/en/latest/)
5. NeuroMatch Academy 的自编码器部分
6. PyTorch 官方文档 (https://pytorch.org/docs/stable/index.html)