线性代数

朱明超

deityrayleigh@gmail.com

1 标量, 向量, 矩阵, 张量

1. **标量** (Scalar): 表示一个单独的**数**，通常用斜体小写字母表示，如 $s \in \mathbb{R}, n \in \mathbb{N}$。

2. **向量** (Vector)：表示**一列数**，这些数有序排列的，可以通过下标获取对应值，通常用粗体小写字母表示：$\boldsymbol{x} \in \mathbb{R}^n$ ，它表示元素取实数，且有 n 个元素，第一个元素表示为：x_1。将向量写成列向量的形式：

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} \tag{1}$$

有时需要向量的子集，例如第 $1, 3, 6$ 个元素，那么我们可以令集合 $S = \{1, 3, 6\}$ ，然后用 $\boldsymbol{x}_S$ 来表示这个子集。另外，我们用符号 $-$ 表示集合的补集：$\boldsymbol{x}_{-1}$ 表示除 x_1 外 $\boldsymbol{x}$ 中的所有元素，$\boldsymbol{x}_{-S}$ 表示除 x_1, x_3, x_6 外 $\boldsymbol{x}$ 中的所有元素。

3. **矩阵** (Matrix): 表示一个**二维数组**，每个元素的下标由两个数字确定，通常用大写粗体字母表示：$\boldsymbol{A} \in \mathbb{R}^{m \times n}$ ，它表示元素取实数的 m 行 n 列矩阵，其元素可以表示为：$\boldsymbol{A}_{1,1}, \boldsymbol{A}_{m,n}$ 。我们用 $:$ 表示矩阵的一行或者一列：$\boldsymbol{A}_{i,:}$ 为第 i 行，$\boldsymbol{A}_{:,j}$ 为第 j 列。

矩阵可以写成这样的形式：

$$\begin{bmatrix} \boldsymbol{A}_{1,1} & \boldsymbol{A}_{1,2} \\ \boldsymbol{A}_{2,1} & \boldsymbol{A}_{2,2} \end{bmatrix} \tag{2}$$

有时我们需要对矩阵进行**逐元素操作**，如将函数 f 应用到 $\boldsymbol{A}$ 的所有元素上，此时我们用 $f(\boldsymbol{A})_{i,j}$ 表示。

4. **张量** (Tensor): **超过二维的数组**，我们用 A 表示张量，$\mathsf{A}_{i,j,k}$ 表示其元素（三维张量情况下）。

[1]:
```
import numpy as np
```

[2]:
```
# 标量
s = 5
# 向量
v = np.array([1,2])
# 矩阵
m = np.array([[1,2], [3,4]])
# 张量
t = np.array([
  [[1,2,3],[4,5,6],[7,8,9]],
  [[11,12,13],[14,15,16],[17,18,19]],
  [[21,22,23],[24,25,26],[27,28,29]],
  ])
print("标量: " + str(s))
print("向量: " + str(v))
print("矩阵: " + str(m))
print("张量: " + str(t))
```

```
标量: 5
向量: [1 2]
矩阵: [[1 2]
 [3 4]]
张量: [[[ 1  2  3]
  [ 4  5  6]
  [ 7  8  9]]
 [[11 12 13]
  [14 15 16]
  [17 18 19]]
 [[21 22 23]
```

```
[24 25 26]
[27 28 29]]]
```

2 矩阵转置

矩阵转置 (Transpose) 相当于**沿着对角线翻转**，定义如下：

$$\boldsymbol{A}_{i,j}^{\top} = \boldsymbol{A}_{i,j} \tag{3}$$

矩阵转置的转置等于矩阵本身：

$$\left(A^{\top}\right)^{\top} = A \tag{4}$$

转置将矩阵的形状从 $m \times n$ 变成了 $n \times m$ 。

向量可以看成是**只有一列的矩阵**，为了方便，我们可以使用行向量加转置的操作，如：$\boldsymbol{x} = [x_1, x_2, x_3]^{\top}$。

标量也可以看成是**一行一列的矩阵**，其转置等于它自身：$a^{\top} = a$ 。

[3]:
```
A = np.array([[1.0,2.0],[1.0,0.0],[2.0,3.0]])
A_t = A.transpose()
print("A:", A)
print("A 的转置:", A_t)
```

```
A: [[1. 2.]
 [1. 0.]
 [2. 3.]]
A 的转置: [[1. 1. 2.]
 [2. 0. 3.]]
```

3 矩阵加法

加法即**对应元素相加**，要求两个矩阵的形状一样：

$$\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}, C_{i,j} = A_{i,j} + B_{i,j} \tag{5}$$

数乘即**一个标量**与**矩阵每个元素相乘**：

$$\boldsymbol{D} = a \cdot \boldsymbol{B} + c, D_{i,j} = a \cdot B_{i,j} + c \tag{6}$$

有时我们允许矩阵和向量相加的，得到一个矩阵，把 $\boldsymbol{b}$ 加到了 A 的每一行上，本质上是构造了一个将 $\boldsymbol{b}$ 按行复制的一个新矩阵，这种机制叫做广播 (Broadcasting)：

$$\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{b}, C_{i,j} = A_{i,j} + b_j \tag{7}$$

[4]:
```
a = np.array([[1.0,2.0],[3.0,4.0]])
b = np.array([[6.0,7.0],[8.0,9.0]])
print("矩阵相加：", a + b)
```

```
矩阵相加： [[ 7.  9.]
 [11. 13.]]
```

4 矩阵乘法

两个矩阵相乘得到第三个矩阵，我们需要 $\boldsymbol{A}$ 的形状为 $m \times n$，$\boldsymbol{B}$ 的形状为 $n \times p$，得到的矩阵为 $\boldsymbol{C}$ 的形状为 $m \times p$：

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B} \tag{8}$$

具体定义为

$$C_{i,j} = \sum_{k} A_{i,k} B_{k,j} \tag{9}$$

注意矩阵乘法不是元素对应相乘，元素对应相乘又叫 Hadamard 乘积，记作 $\boldsymbol{A} \odot \boldsymbol{B}$。

向量可以看作是列为 1 的矩阵，两个相同维数的向量 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 的点乘（Dot Product）或者内积，可以表示为 $\boldsymbol{x}^{\top}\boldsymbol{y}$。

我们也可以把矩阵乘法理解为：$C_{i,j}$ 表示 $\boldsymbol{A}$ 的第 i 行与 $\boldsymbol{B}$ 的第 j 列的点积。

[5]:
```
m1 = np.array([[1.0,3.0],[1.0,0.0]])
m2 = np.array([[1.0,2.0],[5.0,0.0]])
print("按矩阵乘法规则：", np.dot(m1, m2))
print("按逐元素相乘：", np.multiply(m1, m2))
print("按逐元素相乘：", m1*m2)
v1 = np.array([1.0,2.0])
v2 = np.array([4.0,5.0])
print("向量内积：", np.dot(v1, v2))
```

```
按矩阵乘法规则： [[16.  2.]
 [ 1.  2.]]
按逐元素相乘： [[1. 6.]
 [5. 0.]]
按逐元素相乘： [[1. 6.]
 [5. 0.]]
向量内积： 14.0
```

5 单位矩阵

为了引入矩阵的逆，我们需要先定义单位矩阵 (Identity Matrix)：单位矩阵乘以任意一个向量等于这个向量本身。记 $\boldsymbol{I}_n$ 为保持 n 维向量不变的单位矩阵，即：

$$\boldsymbol{I}_n \in \mathbb{R}^{n\times n}, \forall \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{I}_n\boldsymbol{x} = \boldsymbol{x} \tag{10}$$

单位矩阵的结构十分简单，所有的对角元素都为 1 ，其他元素都为 0 ，如：

$$\boldsymbol{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{11}$$

[6]:
```
np.identity(3)
```

[6]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

6 矩阵的逆

矩阵 $\boldsymbol{A}$ 的逆 (Inversion) 记作 $\boldsymbol{A}^{-1}$ ，定义为一个矩阵使得

$$\boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}_n \tag{12}$$

如果 $\boldsymbol{A}^{-1}$ 存在，那么线性方程组 $\boldsymbol{A}x = b$ 的解为：

$$\boldsymbol{A}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{I}_n\boldsymbol{x} = \boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b} \tag{13}$$

[7]:
```
A = [[1.0,2.0],[3.0,4.0]]
A_inv = np.linalg.inv(A)
print("A 的逆矩阵", A_inv)
```

```
A 的逆矩阵 [[-2.   1. ]
 [ 1.5 -0.5]]
```

7 范数

通常我们用范数 (norm) 来衡量向量，向量的 L^p 范数定义为：

$$\|\boldsymbol{x}\|_p = \left(\sum_i |x_i|^p\right)^{\frac{1}{p}}, p \in \mathbb{R}, p \geq 1 \tag{14}$$

L^2 范数，也称欧几里得范数 (Euclidean norm)，是**向量 $\boldsymbol{x}$ 到原点的欧几里得距离**。有时也用 L^2 范数的平方来衡量向量：$\boldsymbol{x}^\top\boldsymbol{x}$ 。事实上，平方 L^2 范数在计算上更为便利，例如它的对 $\boldsymbol{x}$ 梯度的各个分量只依赖于 $\boldsymbol{x}$ 的对应的各个分量，而 L^2 范数对 $\boldsymbol{x}$ 梯度的各个分量要依赖于整个 $\boldsymbol{x}$ 向量。

L^1 范数：L^2 范数并不一定适用于所有的情况，它在原点附近的增长就十分缓慢，因此不适用于需要区别 0 和非常小但是非 0 值的情况。L^1 范数就是一个比较好的选择，它在所有方向上的增长速率都是一样的，定义为：

$$\|\boldsymbol{x}\|_1 = \sum_i |x_i| \tag{15}$$

它经常使用在需要**区分 0 和非 0 元素**的情形中。

L^0 范数：如果需要衡量向量中非 0 元素的个数，但**它并不是一个范数** (不满足三角不等式和数乘)，此时 L^1 范数可以作为它的一个替代。

L^∞ 范数：它在数学上是向量元素绝对值的最大值，因此也被叫做 (`Max norm`)：

$$\|\boldsymbol{x}\|_\infty = \max_i |x_i| \tag{16}$$

有时我们想衡量一个矩阵，机器学习中通常使用的是 `F` 范数 (`Frobenius norm`)，其定义为：

$$\|\boldsymbol{A}\|_F = \sqrt{\sum_{i,j} A_{i,j}^2} \tag{17}$$

[8]:
```
a = np.array([1.0,3.0])
print("向量 2 范数", np.linalg.norm(a,ord=2))
print("向量 1 范数", np.linalg.norm(a,ord=1))
print("向量无穷范数", np.linalg.norm(a,ord=np.inf))
```

```
向量 2 范数 3.1622776601683795
向量 1 范数 4.0
向量无穷范数 3.0
```

[9]:
```
a = np.array([[1.0,3.0],[2.0,1.0]])
print("矩阵 F 范数", np.linalg.norm(a,ord="fro"))
```

```
矩阵 F 范数 3.872983346207417
```

8 特征值分解

如果一个 $n \times n$ 矩阵 $\boldsymbol{A}$ 有 n 组线性无关的单位特征向量 $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}\}$ ，以及对应的特征值 $\lambda_1, \ldots, \lambda_n$ 。将这些特征向量按列拼接成一个矩阵：$\boldsymbol{V} = [\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}]$ ，并将对应的特征值拼接成一个向量：$\boldsymbol{\lambda} = [\lambda_1, \ldots, \lambda_n]$ 。

$\boldsymbol{A}$ 的特征值分解 (`Eigendecomposition`) 为：

$$\boldsymbol{A} = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1} \tag{18}$$

注意：

- 不是所有的矩阵都有特征值分解
- 在某些情况下，实矩阵的特征值分解可能会得到复矩阵

[10]:
```
A = np.array([[1.0,2.0,3.0],
              [4.0,5.0,6.0],
              [7.0,8.0,9.0]])
# 计算特征值
print("特征值:", np.linalg.eigvals(A))
# 计算特征值和特征向量
eigvals,eigvectors = np.linalg.eig(A)
print("特征值:", eigvals)
print("特征向量:", eigvectors)
```

```
特征值: [ 1.61168440e+01 -1.11684397e+00 -3.73313677e-16]
特征值: [ 1.61168440e+01 -1.11684397e+00 -3.73313677e-16]
特征向量: [[-0.23197069 -0.78583024  0.40824829]
 [-0.52532209 -0.08675134 -0.81649658]
 [-0.8186735   0.61232756  0.40824829]]
```

9 奇异值分解

奇异值分解 (Singular Value Decomposition, SVD) 提供了另一种分解矩阵的方式，将其分解为奇异向量和奇异值。

与特征值分解相比，奇异值分解更加通用，所有的实矩阵都可以进行奇异值分解，而特征值分解只对某些方阵可以。

奇异值分解的形式为：

$$\boldsymbol{A}=\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\top} \tag{19}$$

若 $\boldsymbol{A}$ 是 $m\times n$ 的，那么 $\boldsymbol{U}$ 是 $m\times m$ 的，其列向量称为左奇异向量，而 $\boldsymbol{V}$ 是 $n\times n$ 的，其列向量称为右奇异向量，而 $\boldsymbol{\Sigma}$ 是 $m\times n$ 的一个对角矩阵，其对角元素称为矩阵 $\boldsymbol{A}$ 的奇异值。

事实上，左奇异向量是 $\boldsymbol{A}\boldsymbol{A}^{\top}$ 的特征向量，而右奇异向量是 $\boldsymbol{A}^{\top}\boldsymbol{A}$ 的特征向量，非 0 奇异值的平方是 $\boldsymbol{A}^{\top}\boldsymbol{A}$ 的非 0 特征值。

[11]:
```
A = np.array([[1.0,2.0,3.0],
        [4.0,5.0,6.0]])
U,D,V = np.linalg.svd(A)
print("U:", U)
print("D:", D)
print("V:", V)
```

```
U: [[-0.3863177  -0.92236578]
 [-0.92236578  0.3863177 ]]
D: [9.508032   0.77286964]
V: [[-0.42866713 -0.56630692 -0.7039467 ]
 [ 0.80596391  0.11238241 -0.58119908]
 [ 0.40824829 -0.81649658  0.40824829]]
```

10 PCA (主成分分析)

假设我们有 m 个数据点 $\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(m)}\in\mathbb{R}^n$ ，对于每个数据点 $\boldsymbol{x}^{(i)}$ ，我们希望找到一个对应的点 $\boldsymbol{c}^{(i)}\in\mathbb{R}^l, l<n$ 去表示它 (相当于对它进行降维)，并且让损失的信息量尽可能少。

我们可以将这个过程看作是一个编码解码的过程，设编码和解码函数分别为 f,g ，则有 $f(\boldsymbol{x})=\boldsymbol{c},\boldsymbol{x}\approx g(f(\boldsymbol{x}))$。考虑一个线性解码函数 $g(\boldsymbol{c})=\boldsymbol{D}\boldsymbol{c},\boldsymbol{D}\in\mathbb{R}^{n\times l}$ ，为了计算方便，我们将这个矩阵的列向量约束为相互正交的。另一方面，考虑到存在尺度放缩的问题，我们将这个矩阵的列向量约束为具有单位范数来获得唯一解。

对于给定的 $\boldsymbol{x}$ ，我们需要找到信息损失最小的 $\boldsymbol{c}^{\star}$ ，即求解：

$$\boldsymbol{c}^{\star}=\arg\min_{\boldsymbol{c}}\|\boldsymbol{x}-g(\boldsymbol{c})\|_2=\arg\min_{\boldsymbol{c}}\|\boldsymbol{x}-g(\boldsymbol{c})\|_2^2 \tag{20}$$

这里我们用二范数来衡量信息的损失。展开之后我们有：

$$\|\boldsymbol{x}-g(\boldsymbol{c})\|_2^2=(\boldsymbol{x}-g(\boldsymbol{c}))^{\top}(\boldsymbol{x}-g(\boldsymbol{c}))=\boldsymbol{x}^{\top}\boldsymbol{x}-2\boldsymbol{x}^{\top}g(\boldsymbol{c})+g(\boldsymbol{c})^{\top}g(\boldsymbol{c}) \tag{21}$$

结合 $g(\boldsymbol{c})$ 的表达式，忽略不依赖 $\boldsymbol{c}$ 的 $\boldsymbol{x}^{\top}\boldsymbol{x}$ 项，我们有：

$$\begin{aligned}\boldsymbol{c}^{\star}&=\arg\min_{\boldsymbol{c}}-2\boldsymbol{x}^{\top}\boldsymbol{D}\boldsymbol{c}+\boldsymbol{c}^{\top}\boldsymbol{D}^{\top}\boldsymbol{D}\boldsymbol{c}\\&=\arg\min_{\boldsymbol{c}}-2\boldsymbol{x}^{\top}\boldsymbol{D}\boldsymbol{c}+\boldsymbol{c}^{\top}\boldsymbol{I}_l\boldsymbol{c}\\&=\arg\min_{\boldsymbol{c}}-2\boldsymbol{x}^{\top}\boldsymbol{D}\boldsymbol{c}+\boldsymbol{c}^{\top}\boldsymbol{c}\end{aligned} \tag{22}$$

这里 $\boldsymbol{D}$ 具有单位正交性。

对 $\boldsymbol{c}$ 求梯度，并令其为零，我们有：

$$\begin{aligned}\nabla_{\boldsymbol{c}}(-2\boldsymbol{x}^{\top}\boldsymbol{D}\boldsymbol{c}+\boldsymbol{c}^{\top}\boldsymbol{c})&=\boldsymbol{0}\\-2\boldsymbol{D}^{\top}\boldsymbol{x}+2\boldsymbol{c}&=\boldsymbol{0}\\\boldsymbol{c}&=\boldsymbol{D}^{\top}\boldsymbol{x}\end{aligned} \tag{23}$$

因此，我们的编码函数为：

$$f(\boldsymbol{x})=\boldsymbol{D}^{\top}\boldsymbol{x} \tag{24}$$

此时通过编码解码得到的重构为：

$$r(\boldsymbol{x})=g(f(\boldsymbol{x}))=\boldsymbol{D}\boldsymbol{D}^{\top}\boldsymbol{x} \tag{25}$$

接下来求解最优的变换 $\boldsymbol{D}$ 。由于我们需要将 $\boldsymbol{D}$ 应用到所有的 $\boldsymbol{x}_i$ 上，所以我们需要最优化：

$$\begin{aligned} &\boldsymbol{D}^\star = \arg\min_{\boldsymbol{D}} \sqrt{\sum_{i,j}(\boldsymbol{x}_j^{(i)} - r(\boldsymbol{x}^{(i)})_j)^2} \\ &s.t.\ \boldsymbol{D}^\top \boldsymbol{D} = \boldsymbol{I}_l \end{aligned} \tag{26}$$

为了方便，我们考虑 $l = 1$ 的情况，此时问题简化为：

$$\begin{aligned} &\boldsymbol{d}^\star = \arg\min_{\boldsymbol{d}} \sum_{i}(\boldsymbol{x}_j^{(i)} - \boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{x}^{(i)}))^2 \\ &s.t.\ \boldsymbol{d}^\top \boldsymbol{d} = 1 \end{aligned} \tag{27}$$

考虑 F 范数，并进一步的推导：

$$\begin{aligned} &\boldsymbol{d}^\star = \arg\max_{\boldsymbol{d}} \mathrm{Tr}(\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}) \\ &s.t.\ \boldsymbol{d}^\top \boldsymbol{d} = 1 \end{aligned} \tag{28}$$

优化问题可以用特征分解来求解。但实际计算时，我们会采用如下方式计算：

PCA 将输入 $\boldsymbol{x}$ 投影表示成 $\boldsymbol{c}$。$\boldsymbol{c}$ 是比原始输入维数更低的表示，同时使得元素之间线性无关。假设有一个 $m \times n$ 的矩阵 $\boldsymbol{X}$，数据的均值为零，即 $\mathbb{E}[\boldsymbol{x}] = 0$ ，$\boldsymbol{X}$ 对应的无偏样本协方差矩阵：$\mathrm{Var}\,[\boldsymbol{x}] = \frac{1}{m-1}\boldsymbol{X}^\top \boldsymbol{X}$。

PCA 是**通过线性变换找到一个 $\mathbf{Var}\,[\boldsymbol{c}]$ 是对角矩阵**的表示 $\boldsymbol{c} = \boldsymbol{V}^\top \boldsymbol{x}$ ，**矩阵 $\boldsymbol{X}$ 的主成分可以通过奇异值分解 (SVD) 得到**，也就是说主成分是 $\boldsymbol{X}$ 的右奇异向量。假设 $\boldsymbol{V}$ 是 $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ 奇异值分解的右奇异向量，我们得到原来的特征向量方程：

$$\boldsymbol{X}^\top \boldsymbol{X} = (\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top)^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top = \boldsymbol{V}\boldsymbol{\Sigma}^\top \boldsymbol{U}^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top = \boldsymbol{V}\boldsymbol{\Sigma}^2 \boldsymbol{V}^\top \tag{29}$$

因为根据奇异值的定义 $\boldsymbol{U}^\top \boldsymbol{U} = \boldsymbol{I}$ 。因此 $\boldsymbol{X}$ 的方差可以表示为：$\mathrm{Var}\,[\boldsymbol{x}] = \frac{1}{m-1}\boldsymbol{X}^\top \boldsymbol{X} = \frac{1}{m-1}\boldsymbol{V}\boldsymbol{\Sigma}^2 \boldsymbol{V}^\top$ 。

所以 $\boldsymbol{c}$ 的协方差满足：$\mathrm{Var}\,[\boldsymbol{c}] = \frac{1}{m-1}\boldsymbol{C}^\top \boldsymbol{C} = \frac{1}{m-1}\boldsymbol{V}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{V} = \frac{1}{m-1}\boldsymbol{V}^\top \boldsymbol{V}\boldsymbol{\Sigma}^2 \boldsymbol{V}^\top \boldsymbol{V} = \frac{1}{m-1}\boldsymbol{\Sigma}^2$ ，因为根据奇异值定义 $\boldsymbol{V}^\top \boldsymbol{V} = \boldsymbol{I}$ 。$\boldsymbol{c}$ 的协方差是对角的，$\boldsymbol{c}$ 中的元素是彼此无关的。

以 *iris* 数据为例，展示 PCA 的使用。

[12]:
```
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
%matplotlib inline
```

[13]:
```
# 载入数据
iris = load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['label'] = iris.target
df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
df.label.value_counts()
```

[13]:
```
2    50
1    50
0    50
Name: label, dtype: int64
```

[14]:
```
# 查看数据
df.tail()
```

[14]:

	sepal length	sepal width	petal length	petal width	label
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

[15]:
```
# 查看数据
X = df.iloc[:, 0:4]
y = df.iloc[:, 4]
print("查看第一个数据： \n", X.iloc[0, 0:4])
print("查看第一个标签: \n", y.iloc[0])
```

查看第一个数据：

```
 sepal length    5.1
sepal width     3.5
petal length    1.4
petal width     0.2
Name: 0, dtype: float64
```

查看第一个标签：

```
 0
```

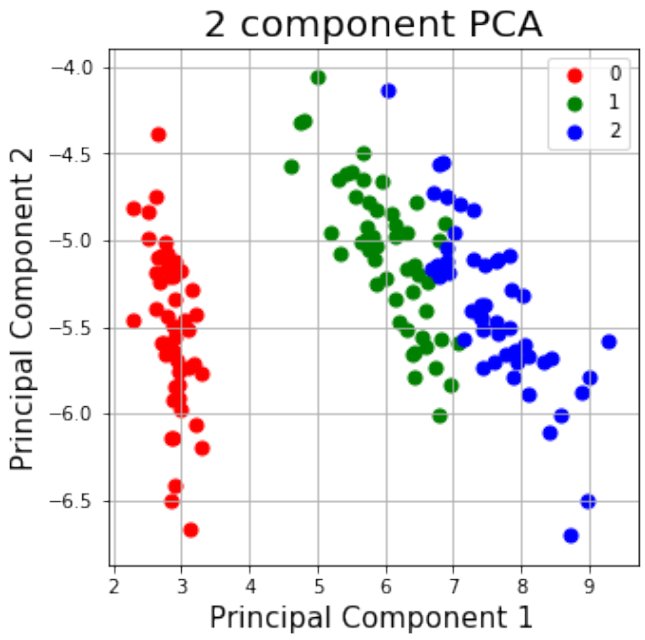
[16]:
```
class PCA():
    def __init__(self):
        pass

    def fit(self, X, n_components):
        n_samples = np.shape(X)[0]
        covariance_matrix = (1 / (n_samples-1)) * (X - X.mean(axis=0)).T.dot(X - X.mean(axis=0))
        # 对协方差矩阵进行特征值分解
        eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)
        # 对特征值（特征向量）从大到小排序
        idx = eigenvalues.argsort()[::-1]
        eigenvalues = eigenvalues[idx][:n_components]
        eigenvectors = np.atleast_1d(eigenvectors[:, idx])[:, :n_components]
        # 得到低维表示
        X_transformed = X.dot(eigenvectors)
        return X_transformed
```

[17]:
```
model = PCA()
Y = model.fit(X, 2)
```

[18]:
```
principalDf = pd.DataFrame(np.array(Y),
                           columns=['principal component 1', 'principal component 2'])
Df = pd.concat([principalDf, y], axis = 1)
fig = plt.figure(figsize = (5,5))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 component PCA', fontsize = 20)

targets = [0, 1, 2]
# ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
colors = ['r', 'g', 'b']
for target, color in zip(targets,colors):
    indicesToKeep = Df['label'] == target
    ax.scatter(Df.loc[indicesToKeep, 'principal component 1']
               , Df.loc[indicesToKeep, 'principal component 2']
               , c = color
               , s = 50)
ax.legend(targets)
ax.grid()
```

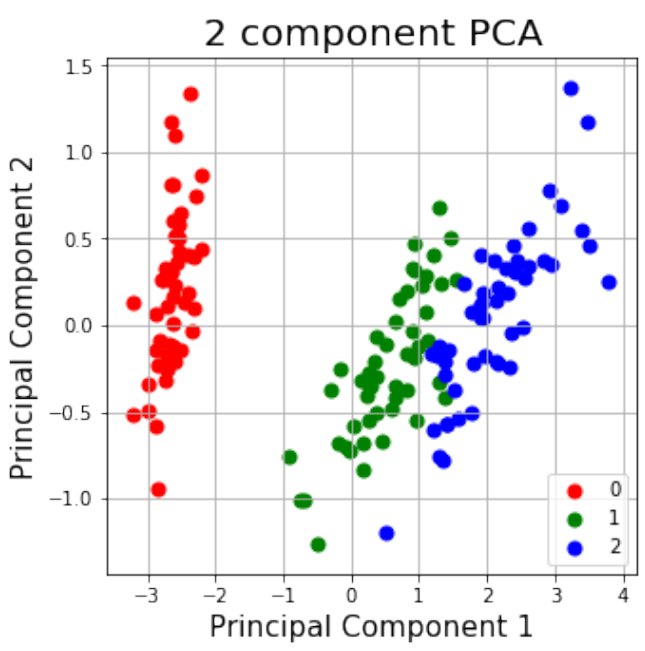


使用 *sklearn* 包实现 PCA

[19]:
```
from sklearn.decomposition import PCA as sklearnPCA
sklearn_pca = sklearnPCA(n_components=2)
Y = sklearn_pca.fit_transform(X)
```

[20]:
```
principalDf = pd.DataFrame(data = np.array(Y), columns = ['principal component 1', 'principal component 2'])
Df = pd.concat([principalDf, y], axis = 1)
fig = plt.figure(figsize = (5,5))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 component PCA', fontsize = 20)

targets = [0, 1, 2]
# ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
colors = ['r', 'g', 'b']
for target, color in zip(targets,colors):
    indicesToKeep = Df['label'] == target
    ax.scatter(Df.loc[indicesToKeep, 'principal component 1']
               , Df.loc[indicesToKeep, 'principal component 2']
               , c = color
               , s = 50)
ax.legend(targets)
ax.grid()
```



[21]:
```
import numpy, pandas, matplotlib, sklearn
print("numpy:", numpy.__version__)
print("pandas:", pandas.__version__)
```

```
print("matplotlib:", matplotlib.__version__)
print("sklearn:", sklearn.__version__)
```

```
numpy: 1.14.5
pandas: 0.25.1
matplotlib: 3.1.1
sklearn: 0.21.3
```