第二章 线性代数

2.1 标量 向量 矩阵 张量

- 标量（Scalar）：一个标量就是一个单独的数。我们用斜体表示标量。标量通常被赋予小写的变量名称。例如：实数标量，令 $s \in R$；自然数标量，令$n \in N$。
- 向量 (Vector)：一个向量是一列数。这些数是有序排列的。通过索引下标确定单独的元素。向量 x 的第一个元素是 x^1，将向量写成列向量的形式：

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

- 矩阵（matrix）：矩阵是一个二维数组，其中每个元素被两个数字确定。例如实数矩阵高度为m，宽度为n，表示为$A \in R^{m \times n}$，元素表示为$A_{1,1}$，$A_{m,n}$。我们用：表示矩阵的一行或者一列:$A_{i,:}$ 为第 i 行，$A_{:,j}$ 为第 j 列。矩阵写成：

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$$

  $f(A)_{i,j}$ 表示函数f 作用在 A 上输出的矩阵的第 i 行第 j 列元素。

- 张量（Tensor）：超过二维的数组，用A表示张量，$A_{i,j,k}$表示三维张量的元素。

```
import numpy as np

# 标量
```

```
s = 3
# 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: 3
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]]]
```

矩阵的转置$A^T_{i,j} = A_{j,i}$，对角线元素对换。

矩阵加法对应元素相加，要求两个矩阵形状一样。

$$C = A + B, C_{i,j} = A_{i,j} + B_{i,j}$$

我们允许矩阵和向量相加得到一个矩阵，把 b 加到了 A 的每一行上，本质上是构造了一个将 b 按行复制的一个新矩阵，这种机制叫做广

播 (Broadcasting):

$$C = A + b, C_{i,j} = A_{i,j} + b_j$$

```
# 矩阵相加
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)

# 矩阵与向量相加，广播
c = np.array([[4.0],[5.0]])
print('广播: ', a + c)
矩阵相加:  [[ 7.  9.]
 [11. 13.]]
广播:  [[5. 6.]
 [8. 9.]]
```

2.2 矩阵乘法

两个矩阵相乘得到第三个矩阵，A 的形状为 m × n，B 的形状为 n × p，得到的矩阵为 C 的形状为 m × p:

$$C = AB$$
$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

矩阵相乘不是对应元素相乘，对应元素相乘是element-wise product 或 Hadamard product。

$$A \odot B$$

两个相同维数的向量 x 和 y 的点乘(Dot Product)或者内积，可以表示为 $x^\top y$。C=AB中计算 $C_{i,j}$作为A的i行和B的j列做dot product。

```
# 矩阵乘法
m1 = np.array([[1.0,3.0],[1.0,0.0]])
m2 = np.array([[1.0,2.0],[5.0,0.0]])
print('矩阵乘法: ', np.dot(m1,m2))
print('逐元素相乘: ', np.multiply(m1, m2))
print('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))
```

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

2.3 单位矩阵和逆矩阵

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。

$$I_n \in R^{n \times n}, \forall x \in R^n, I_n x = x$$

单位矩阵，所有的对角元素都为 1 ，其他元素都为 0，如:

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
# 单位矩阵
np.identity(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])

np.eye(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

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

$$A^{-1}A = I_n$$

如果 A^{-1}存在，线性方程组Ax=b的解为：

$$A^{-1}Ax = I_n x = x = A^{-1}b$$

```
# 矩阵的逆
A = [[1.0,2.0],[3.0,4.0]]
A_inv = np.linalg.inv(A)
print('矩阵A的逆: ', A_inv)

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

2.4 线性相关和生成子空间

对于分析方程组的解，可以理解成，A的列向量看作从原点出发，指向不同的行走方向。这些列向量决定了有多少种方式可以到达b。同样，每个x表示每个方向要走多远。则x_i表示在第i列上走多远：

$$Ax = \sum_i x_i A_{:,i}$$

这种操作叫线性组合（linear combination）。一组向量的线性组合是每个向量$v^{(i)}$乘以扩张系数加起来的结果：

$$\sum_i c_i v^{(i)}$$

一组向量空间（span）是通过线性组合能到达的所有点的集合。

判断Ax=b有解，等同于检测b是否在A的列向量生成子空间（span）中。这个特殊的生成子空间成为A的列空间（column spcae）或者A的值域（range）。A的列空间是整个R^m的要求，意味着A至少有m列，即n>=m。但这只是不要条件（necessary condition），不是充分条件（sufficenti condition）。因为有些列向量是多余的，有些列向量是线性相关的，不能涵盖真实的列向量数目，某些列向量可以消掉，作用是一样的。

这种冗余被称为线性相关（linear dependence），一组向量中，任何一个向量都不向量组的线性组合，称向量组线性无关（linear independent）。矩阵可逆要确保Ax=b最多有一个解，也就是特解。因此确保矩阵最多有m列。综上所述，矩阵必须是方阵，所有的列必须是线性无关的。带有线性相关列向量的方阵成为奇异矩阵（singular）。如果矩阵 A 不是一个方阵或者是一个奇异的方阵，该方程仍然可能有解。但是我们不能使用矩阵逆去求解。

2.5 范数（Norms）

衡量向量的大小，用叫范数的函数。L^p范数

$$\|x\| = (\sum_i |x_i|^p)\frac{1}{p}$$

其中

$$p \in R,\ p \geq 1。$$

向量x的范数是衡量从原点到x的距离。范数是满足下列性质的函数：

$$\bullet f(x) = 0 \Rightarrow x = 0$$
$$\bullet\ f(x+y) \leq f(x) + f(y)$$
$$\bullet\ \forall \alpha \in R, f(\alpha x) = |\alpha| f(x)$$

当p=2，L^2范数被称为 欧几里得范数 （Euclidean norm）。它表示从原点出发到向量x确定的点的欧几里得距离。L^2范数经常简化表示为$\|x\|$略去了下标2。平方L^2范数也经常用来衡量向量大小，可以简单地通过 点积 x^Tx计算。

平方L^2范数计算比L^2范数本身方便。例如，平方L^2范数对x中每个元素的导数只取决于对应元素，而L^2范数导数取决于整个向量。有时候平方L^2范数也不太理想，因为在原点附近增长缓慢。有些机器学习应用中，区分零和非零是很重要的。我们转而去用在所有位置斜率相同的函数。同时保持简单的数学形式的函数：L^1范数：

$$\|x\|_1 = \sum_i |x_i|$$

当零和非零元素之间的差异非常重要时，通常使用L^1范数。

L^1范数也用作L^0范数作为表示非零元素个数的替代者。

最大范数（max norm）是L^∞。表示向量中最大量级的元素的绝对值。

$$\|x\|_\infty = \max_i |x_i|$$

需要测试矩阵的大小，使用Frobenius范数

$$\|x\|_F = \sqrt{\sum_{i,j} A_i^2, j}$$

两个向量的点积（dot product）可以用范数表示。

$$x^T y = \|x\|_2 \|y\|_2 \cos\theta$$

其中 θ 是 x和y的夹角。

```
# 范数
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

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
```

2.6 特殊的矩阵和向量

对角矩阵（diagonal matrix）除主对角线都是0元素，非0元素在主对角线上。我们已经看到一个对角矩阵的例子是单位矩阵。用diag(x)去表示一个对角方阵，对角元素由向量v给定。对角矩阵计算很方便。计算乘法diag(x)，我们只需放大每个元素x_i成v_i倍。换句话说，

$$diag(v)x = v \odot x$$

计算对角方阵的逆阵也很方便。对角方阵的逆存在，当且仅当对角元素都非零。

$$diag(x)^{-1} = diag([1/v_1, \dots, 1/v_n]^T)$$

不是所有的对角阵都是方阵。可以构造一个长方形的对角阵。非方阵的对角阵没有逆阵，但仍然可以高效计算。

对陈（symmetric）矩阵，转置和自身相等的矩阵：

$$A = A^T$$

比如度量距离的矩阵就是对称矩阵，$A_{i,j} = A_{j,i}$。

单位向量（unit vector）是模等于1的向量，是具有 `单位范数` （unit norm）的向量：

$$\|x\|_2 = 1$$

向量x和向量y 正交 （orthogonal）则，$x^Ty=0$。如果两个向量都有非零范数，这两个向量夹角90度。在R^n最多有n个非零范数的向量相互正交。如果这些矩阵不仅正交，并且范数是1，称为 标准正交 （orthonormal）。

一个 正交矩阵 （orthonormal）是一个方阵，他的行相互是标准正交，列也是标准正交。

$$A^TA = AA^T = I$$

也就是

$$A^{-1} = A^T$$

因此正交矩阵很受欢迎是因为计算很方便。

2.7 特征值分解

很多数学对象分解后更好理解，或找到通用的属性。可以分解整数为质数，我们也能分解矩阵发现一些不明显的特征。矩阵分解常用是 特征值分解 （eigendecomposition），把矩阵分解成一组特征向量和特征值。

方阵A的特征向量（eigenvector）是非零向量v乘A相当于对v进行缩放。

$$Av = \lambda v$$

标量λ被称为这个特征向量对应的 特征值 （eigenvalue）。特征向量v和乘以一个标量s后的sv都是A的特征向量，并且v和sv有相同的特征值。因此只考虑单位特征向量。

假设矩阵A有n个线性无关的特征向量$\{v^{(1)}, \dots, v^{(n)}\}$，对应着特征值$\{\lambda_1, \dots, \lambda_n\}$，我们将特征向量拼接成一个矩阵V，每一列是一个特征向量：$\{v^{(1)}, \dots, v^{(n)}\}$。同样，我们拼接特征值成一个向量$\lambda = [\lambda_1, \dots, \lambda_n]$，所以A的特征值分解（eigendecomposition）写成：

$$A = Vdiag(\lambda)V^{-1}$$

```
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]]
```

我们经常做矩阵分解成特征值和特征向量，这样可以帮助我们分析矩阵的特性。就像质因素分解帮组我们理解整数一样。

不是每个矩阵都可以分解成特征值和特征向量。某些情况下，特征分解会涉及复数而非实数。具体来说，每个实对称矩阵（real symmetric）都可以分解成实特征向量和实特征值表达式：

$$A = Q\Lambda Q^T$$

Q是A的特征向量组成的正交矩阵，Λ 是对角矩阵。特征值$\Lambda_{i,i}$与特征向量Q的第i列相关联。记作$Q_{:,i}$。因为Q是正交矩阵，我们可以认为A在$v^{(i)}$方向上伸缩λ_i倍。矩阵特征值分解很有用，如果任意一个特征值是0时，矩阵是奇异（singular）矩阵。实对称矩阵的特征值分解也用来优化二次方程

$$f(x) = x^T Ax, \text{限制} ||x||_2 = 1$$

当x等于A的某个特征向量时，f将返回对应的特征值。在限制条件下，函数f的最大值是最大特征值，最小值是最小特征值。

所有特征值都是正数的矩阵称为 正定 （positive definite），所有特征值都是非负的矩阵成为 半正定 （positive semidefinite）。半正定受关注是因为$\forall x, x^T Ax \geq 0$。此外，正定矩阵还保证$x^T Ax = 0 \Rightarrow x = 0$。

2.8 奇异值分解

除了将矩阵分解成特征值（eigenvalues）和特征向量（eigenvectors）。 奇异值分解 （singular value decomposition SVD）提供了另一种分解矩阵为 奇异值 （singular value）和 奇异向量 （singular vector）的方法。通过奇异值分解我们会得到和特征分解相同类型的信息。然而奇异值分解更广泛，每个实数矩阵都有奇异值分解，但不一定有特征值分解。非方阵没有特征值分解，就必须用奇异值分解。

用特征值分解去分析矩阵A时，得到特征向量构成的矩阵V和特征值构成的向量λ，A可以写作：

$$A = Vdiag(\lambda)V^{-1}$$

奇异值分解类似，只是A由3个矩阵产生：

$$A = UDV^T$$

假设A是一个m×n，那么U是一个m×m的矩阵，D是一个m×n的矩阵，V是一个n×n的矩阵。矩阵 U 和 V 都定义为正交矩阵，而矩阵 D 定义为对角矩阵。注意，矩阵 D 不一定是方阵。对角矩阵对角线上的元素被称为 奇异值 （singular value）。矩阵U的列向量成为 左奇异向量 （left-singular vectors）。矩阵V的列向量被称为 有奇异向量 （right-singular vectors）。

可以用A特征分解去解释奇异值分解。A的 左奇异向量 （left-singular vectors）是AA^T的特征向量。A的 右奇异向量 （right-singular vectors）是A^TA的特征向量。A 的非零奇异值是 A^TA 特征值的平方根，同时也是 $AA^\top$ 特征值的平方根。

（个人理解：左奇异向量U是m×m矩阵，AA^T也是m×m矩阵，因此矩阵形状对上了，右奇异值同理，n×n矩阵。）

```
# 奇异值分解
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]]
```

2.9 Moore-Penrose伪逆

非方矩阵没有逆阵定义。比如想通过矩阵A左逆B来求解线性方程，

$$Ax = y$$

等式左边乘左逆B后，得到

$$x = By$$

如果矩阵A行数大于列数，方程无解。如果矩阵A行数小于列数，有多个解。

A矩阵的伪逆定义为：

$$A^+ = \lim_{a \to 0}(A^T A + \alpha I)^{-1} A^T$$

但是计算伪逆用下面公式：

$$A^+ = V D^+ U^T$$

其中，矩阵 U，D 和 V 是矩阵 A奇异值分解后得到的矩阵。对角矩阵 D 的伪逆 D^+ 是其非零元素取倒数之后再转置得到的。当矩阵 A 的列数多于行数时，使用伪逆求解线性方程是众多可能解法中的一 种。特别地，x = A+y 是方程所有可行解中欧几里得范数 $||x||_2$ 最小的一个。当矩阵 A 的行数多于列数时，可能没有解。在这种情况下，通过伪逆得到的 x 使得 Ax 和 y 的欧几里得距离 $||Ax - y||_2$ 最小。

2.10 迹运算

迹运算返回矩阵对角元素和：

$$Tr(A) = \sum_i A_{i,j}$$

迹运算可以描述 `Frobenius范数` ：

$$||A||_F = \sqrt{Tr(AA^T)}$$

迹运算在转置操作下不变：

$$Tr(A) = Tr(A^T)$$

矩阵相乘，挪动位置后仍然有定义，迹运算不变：

$$Tr(ABC) = Tr(CAB) = Tr(BCA)$$

另一个有用的事实是标量在迹运算后仍然是它自己: $a = Tr(a)$。

2.11 行列式（The Determinate）

行列式，记作$det(A)$，是将方阵A映射到实数的函数，行列式等于矩阵特征值的乘积。行列式的绝对值可以用来衡量矩阵参与矩阵乘法后空间扩大或者缩小 了多少。如果行列式是 0，那么空间至少沿着某一维完全收缩了，使其失去了所有的 体积。如果行列式是 1，那么这个转换保持空间体积不变。

2.12 主成分分析（Principal Components Analysis）

假设在R^n中有m个点$\{x^{(1)},\dots,x^m\}$，假设我想有损压缩这些点，意味着存储这些点用更少的内存，但是会尽量少的损失精度。一种方式使用低纬表示他们。对于每个点$x^{(i)} \in R^n$找到一个对应的编码向量$c^{(i)} \in R^l$。l比n小，比原始数据用更少的内存存储。要找到编码函数对输入生成编码，$f(x)=c$，把编码用解码函数重新生成输入，$x \approx g(f(x))$。

用$g(c)=Dc, where D \in R^{n\times l}$是定义的解码矩阵。为计算方便，PCA限制矩阵D的向量彼此正交，除非$l=n$否则D不是一个正交矩阵。目前所述会有很多解，因为扩张$D_{:,i}$按比例缩小c_i。为了使问题有唯一解，限制D所有列向量有`单位范数`。

要明确如何把每个输入x得到的一个最优编码c^*，一种方式是最小化，输入x和$g(c^*)$之间的距离。用L^2范数衡量距离。

$$c^* = arg\min_c \|x-g(c)\|_2$$

用平方L^2范数替代L^2范数

$$c^* = arg\min_c \|x-g(c)\|_2^2$$

最小化函数可以化简：

$$\begin{aligned}
&(x-g(x))^T(x-g(c)) \\
&= x^Tx - x^Tg(c) - g(c)^Tx + g(c)^Tg(c) \\
&= x^Tx - 2x^Tg(c) + g(c)^Tg(c) && \text{由于标量}g(c)^Tx\text{的转置等于自己}
\end{aligned}$$

由于第一项不依赖c，忽略。优化函数为：

$$\begin{aligned}
c^* &= arg\min_c -2x^Tg(c) + g(c)^Tg(c) \\
c^* &= arg\min_c -2x^TDc + c^TD^TDc && \text{用}g(c)\text{定义替换} \\
&= arg\min_c -2x^TDc + c^TI_lc && \text{阵}D\text{的正交性和单位范数约束} \\
&= arg\min_c -2x^TDc + c^Tc
\end{aligned}$$

用微积分求解最优化问题：

$$\begin{gathered}
\bigtriangledown_c(-2x^TDc + c^Tc) = 0 \\
-2D^Tx + 2c = 0 \\
c = D^Tx
\end{gathered}$$

最优编码x只需要一个矩阵向量相乘，编码向量，编码函数是：

$$f(x) = D^T x$$

也可以定义PCA重构操作：

$$r(x) = g(f(x)) = DD^T x$$

挑选编码矩阵D，回顾下目的是最小化输入和重构之间的L^2距离。要最小化所有维数和所有点误差矩阵Frobenius范数：

$$D^* = arg\min_D \sqrt{\sum_{i,j}(x_j^{(i)} - r(x^{(i)})_j)^2}, subject\ to\ D^T D = I_l$$

为了推导简单令$l = 1$，此时D是个单一向量d。

$$d^* = arg\min_d \sum_i \|x^{(i)} - dd^T x^{(i)}\|_2^2, subject\ to\ \|d\|_2 = 1$$

因为d是$n \times 1$的向量，$d^T x^{(i)}$是标量。标量写在左边：

$$d^* = arg\min_d \sum_i \|x^{(i)} - d^T x^{(i)} d\|_2^2, subject\ to\ \|d\|_2 = 1$$

标量转置不变：

$$d^* = arg\min_d \sum_i \|x^{(i)} - x^{(i)T} dd\|_2^2, subject\ to\ \|d\|_2 = 1$$

把求和写成矩阵形式。$X \in R^{m \times n}$作为向量堆叠地起来的矩阵。$X_{i,:} = x^{(i)T}$。$d \times d^T$得到矩阵是$n \times n$的。

$$d^* = arg\min_d \|X - Xdd^T\|_F^2, subject\ to\ d^T d = 1$$

不考虑限制，化简：

$$
\begin{aligned}
d^* &= arg\min_d \|X - Xdd^T\|_F^2 \\
&= arg\min_d Tr((X - Xdd^T)^T(X - Xdd^T)) \\
&= arg\min_d Tr(X^TX - X^TXdd^T - dd^TX^TX + dd^TX^TXdd^T) \\
&= arg\min_d Tr(X^TX) - Tr(X^TXdd^T) - Tr(dd^TX^TX) + Tr(dd^TXX^Tdd^T) \\
&= arg\min_d -Tr(X^TXdd^T) - Tr(dd^TX^TX) + Tr(dd^TX^TXdd^T) && \text{去掉没有}d\text{的} \\
&= arg\min_d -2Tr(X^TXdd^T) + Tr(dd^TX^TXdd^T) && \text{矩阵相乘顺序改变迹不变} \\
&= arg\min_d -2Tr(X^TXdd^T) + Tr(X^TXdd^Tdd^T) && \text{矩阵相乘顺序改变迹不变}
\end{aligned}
$$

在考虑约束条件:

$$
\begin{aligned}
& arg\min_d -2Tr(X^TXdd^T) + Tr(X^TXdd^Tdd^T), subject\ to\ d^Td = 1 \\
=& arg\min_d -2Tr(X^TXdd^T) + Tr(X^TXdd^T), subject\ to\ d^Td = 1 \\
=& arg\min_d -Tr(X^TXdd^T), subject\ to\ d^Td = 1 \\
=& arg\max_d Tr(X^TXdd^T), subject\ to\ d^Td = 1 \\
=& arg\max_d Tr(d^TX^TXd), subject\ to\ d^Td = 1
\end{aligned}
$$

这个优化问题可以通过特征值分解来求解。最优的d是X^TX最大特征值对应的特征向量。以上$l = 1$仅得到第一个主成分。当$l > 1$，矩阵D是前l个特征值对应的特征向量组成。

```
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

# load data
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()
```

```
2    50
1    50
0    50
Name: label, dtype: int64
```

```
# show data
df.tail()
```

	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

```
df.head()
```

	sepal length	sepal width	petal length	petal width	label
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

```
# show data
X = df.iloc[:, 0:4]
y = df.iloc[:, 4]
print("show first data:\n", X.iloc[0, 0:4])
print("show first label:\n", y.iloc[0])
print(X.shape)

show first data:
 sepal length    5.1
sepal width     3.5
```

```
petal length    1.4
petal width     0.2
Name: 0, dtype: float64
show first label:
 0
(150, 4)
```

```
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))
        print(covariance_matrix.shape)
        # pca covariance matrix
        eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)
        print(eigenvalues.shape, eigenvectors.shape)
        print(eigenvalues, eigenvectors)
        # eigenvalues sort
        idx = eigenvalues.argsort()[::-1]
        print(idx)
        eigenvectors = np.atleast_1d(eigenvectors[:, idx])[:,
:n_components]
        # little dimensions
        X_transformed = X.dot(eigenvectors)
        print(X_transformed.shape)
        return X_transformed
```

解释下这段代码，先通俗的解释下pca。参考博客（https://blog.csdn.net/a10767891/article/details/80288463）

pca主要步骤：

1.输入样本矩阵$m \times n$。

$$X=\begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \dots & \dots & \dots & \dots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{bmatrix}_{m\times n}$$

2.对样本矩阵进行中心化（取均值）

$$X=\begin{bmatrix} x_{11}-\mu_1 & x_{12}-\mu & \dots & x_{1n}-\mu_n \\ x_{21}-\mu_1 & x_{22}-\mu & \dots & x_{2n}-\mu_n \\ \dots & \dots & \dots & \dots \\ x_{m1}-\mu_1 & x_{m2}-\mu & \dots & x_{mn}-\mu_n \end{bmatrix}_{m\times n}$$

3.计算协方差矩阵

$$C=\begin{bmatrix} Cov(x_1,x_1) & Cov(x_1,x_2) & \dots & Cov(x_1,x_n) \\ Cov(x_2,x_1) & Cov(x_2,x_2) & \dots & Cov(x_2,x_n) \\ \dots & \dots & \dots & \dots \\ Cov(x_n,x_1) & Cov(x_n,x_2) & \dots & Cov(x_n,x_n) \end{bmatrix}_{n\times n}$$

简易公式：$C=\frac{1}{m}X^TX$

4.计算协方差矩阵的特征值并取出最大的k个特征值所对应的特征向量，构成新的矩阵。若是用python3的话，借助numpy.linalg.eig函数即可。

5.这个矩阵就是我们要求的特征矩阵，里面每一列就为样本的一维主成分。把样本矩阵投影到以该矩阵为基的新空间中，便可以将n维数据降低成k维数据。

pca关键就是求出协方差矩阵的特征向量矩阵。pca就是降维技术。设有样本$m\times n$，有一个基矩阵P大小为$n\times k(k<n)$，P就是上文提到的编码矩阵$D\in R^{n\times l}$。

令$Y=XP$，则Y矩阵的大小为$m\times k$。这样就把m个样本的维度从n减小到了k。维数减少了，虽然可以大大减少算法的计算量，但是若对基矩阵P选择不当的话就很有可能会导致信息量的缺失。因此我们要选择哪k个基才能保证降维后能最大程度保留原有的信息，是进行设计的主方向。

我们可以知道信息来源于未知。也就是说如果不同样本的同一维度的值差异特别大，那该维度带给我们的信息量就是极大的。转换成数学语言，也就是说某维度的方差越大，它的信息量越大。

PCA算法的优化目标就是: ① 降维后同一维度的方差最大

② 不同维度之间的相关性为0(其实就是让基向量互相正交)。

同一元素的协方差就表示该元素的方差，不同元素之间的协方差就表示它们的相关性。因此这两个优化目标可以用协方差矩阵来表示。

$$C=\begin{bmatrix} Cov(x_1,x_1) & Cov(x_1,x_2) & \dots & Cov(x_1,x_n) \\ Cov(x_2,x_1) & Cov(x_2,x_2) & \dots & Cov(x_2,x_n) \\ \dots & \dots & \dots & \dots \\ Cov(x_n,x_1) & Cov(x_n,x_2) & \dots & Cov(x_n,x_n) \end{bmatrix}_{n\times n} \Rightarrow C_Y=\begin{bmatrix} \delta_{11} & 0 & \dots & 0 \\ 0 & \delta_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \delta_{nn} \end{bmatrix}_{n\times n}$$

样本降维后希望得到的最理想的协方差矩阵（包含最多的信息）。

优化目标是令C_Y矩阵的对角线元素之和最大。即$\max tr(C_Y)$

下面总结下:

设X为$m\times n$样本中心化矩阵，P为$n\times R$的基矩阵（每一列为一个基向量），$Y=XP$，是降维后的样本矩阵。则Y矩阵的协方差:

$$\begin{aligned} C_Y^{'} =& \frac{1}{m}(XP)^T(XP) \\ C_Y^{'} =& P^T\left(\frac{1}{m}X^TX\right)P \end{aligned}$$

$\therefore C_Y^{'}=P^TCP$ （C就是样本的协方差矩阵）

$\because P$是正交 （正交能减少降维后不同维度的相关性）

$\therefore P^TP=I$

所以优化目标表示为:

$$\begin{cases} \max(tr(P^TCP)) \\ P^TP=I \end{cases}$$

根据拉格朗日乘子法得: $f(P)=tr(P^TCP)+\lambda(P^TP-I)$

对$f(P)$求导，取其零点。

补充矩阵求导:

$$\frac{\partial tr(AB)}{\partial A} = B^T$$

$$\frac{\partial X^T X}{\partial X} = X$$

$$\therefore \frac{\partial f}{\partial P} = \frac{\partial tr(P^T CP)}{\partial P} + \lambda \frac{\partial (P^T P)}{\partial P} = \frac{\partial tr(PP^T C)}{\partial P} + \lambda P = C^T P + \lambda P = CP + \lambda P$$

当$\frac{\partial f}{\partial P} = 0$时$CP + \lambda P = 0 \Rightarrow CP = (-\lambda)P$

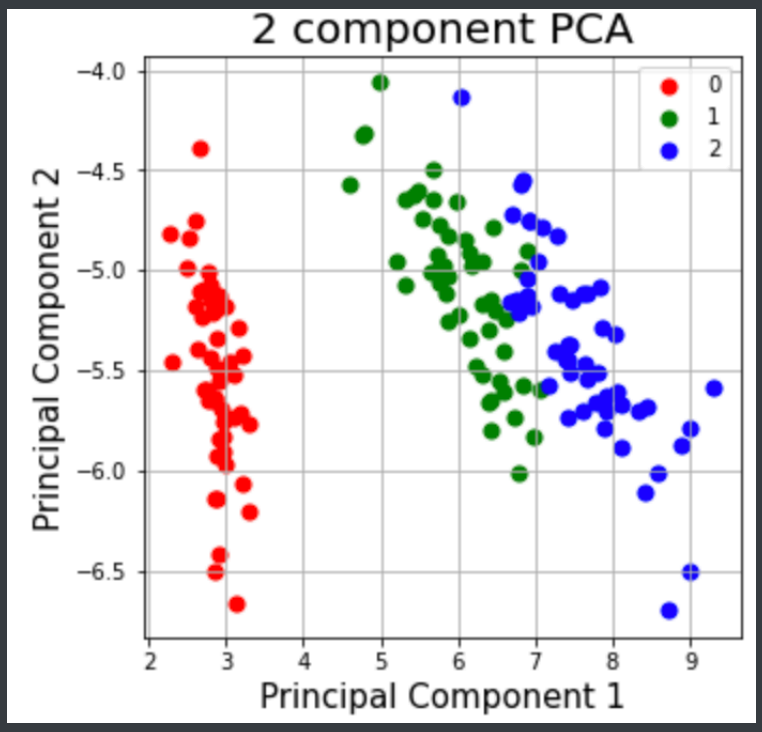
因此又C矩阵的特征向量所构成的基矩阵，就是需要求解的变换矩阵。由该矩阵降维得到的新样本矩阵可以最大程度保留原样本的信息。因此信息量保存能力最大的基向量P，一定是样本矩阵X的协方差矩阵的特征向量。这个推导过程就解释了为什么PCA算法要利用样本协方差的特征向量矩阵来降维。

```
model = PCA()
Y = model.fit(X, 2)

(4, 4)
(4,) (4, 4)
[4.22824171 0.24267075 0.0782095  0.02383509] [[ 0.36138659 -0.65658877
-0.58202985  0.31548719]
 [-0.08452251 -0.73016143  0.59791083 -0.3197231 ]
 [ 0.85667061  0.17337266  0.07623608 -0.47983899]
 [ 0.3582892   0.07548102  0.54583143  0.75365743]]
[0 1 2 3]
(150, 2)
```

```
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()
```



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

numpy: 1.18.5
pandas: 1.1.3

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

matplotlib: 3.3.2
sklearn: 0.23.2
```