1. 数学基础

①线性代数　②概率统计　③优化理论　④图论

2. 代码部分

①NumPy　②Pandas　③SciPy　④Matplotlib

数学基础

一、线性代数

1.哈达玛积

对同型矩阵 $A=(a_{ij})_{m\times n}, B=(b_{ij})_{m\times n}$，有 $C=A\odot B=(c_{ij})_{m\times n}$，其中 $c_{ij}=a_{ij}\times b_{ij}$

如 $A=\begin{pmatrix}1 & 3 & -2\\ 4 & 2 & 5\end{pmatrix}, B=\begin{pmatrix}-2 & 7 & 0\\ 4 & -3 & 1\end{pmatrix}, A\odot B=\begin{pmatrix}-2 & 21 & 0\\ 16 & -6 & 5\end{pmatrix}$

2.克罗内克积

对任意大小的矩阵 $A_{m\times n}, B_{p\times q}$，有 $A\otimes B=\begin{pmatrix}a_{11}B & \cdots & a_{1n}B\\ \vdots & \ddots & \vdots\\ a_{m1}B & \cdots & a_{mn}B\end{pmatrix}$

如 $A=\begin{pmatrix}1 & 2\\ 3 & 1\end{pmatrix}$，$B=\begin{pmatrix}2 & 0\\ 4 & 1\end{pmatrix}, A\otimes B=\begin{pmatrix}1\cdot 2 & 1\cdot 0 & 2\cdot 2 & 2\cdot 0\\ 1\cdot 4 & 1\cdot 1 & 2\cdot 4 & 2\cdot 1\\ 3\cdot 2 & 3\cdot 0 & 1\cdot 2 & 1\cdot 0\\ 3\cdot 4 & 3\cdot 1 & 1\cdot 4 & 1\cdot 1\end{pmatrix}=\begin{pmatrix}2 & 0 & 4 & 0\\ 4 & 1 & 8 & 2\\ 6 & 0 & 2 & 0\\ 12 & 3 & 4 & 1\end{pmatrix}$

3.正定矩阵

若对任意向量 $\vec{x}=(x_1,\cdots,x_n)^T\neq 0$，对称矩阵 A 均有 $x^TAx>0$，则称 A 为正定矩阵。若 $x^TAx\geq 0$，则称 A 为半正定矩阵。若 $x^TAx<0$，则称 A 为负定矩阵。若 $x^TAx\leq 0$，则称 A 为半负定矩阵。

性质：正定矩阵的特征值全为正数，半正定矩阵的特征值全为非负数。

4.正交矩阵

满足 $A^T=A^{-1}$ 或 $A^TA=AA^T=E$ 的矩阵 A。

如 $A=\begin{pmatrix}\frac{1}{2} & -\frac{\sqrt{3}}{2}\\ \frac{\sqrt{3}}{2} & \frac{1}{2}\end{pmatrix}$，则 $A^T=\begin{pmatrix}\frac{1}{2} & \frac{\sqrt{3}}{2}\\ -\frac{\sqrt{3}}{2} & \frac{1}{2}\end{pmatrix}, AA^T=\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}=E_2$

5.矩阵的迹

n 阶**方阵**主对角线上的元素之和 $\mathrm{tr}(A)$

6.矩阵的奇异值

设特征值 λ_i，称 $\sigma_i=\sqrt{\lambda_i}(i=1,2,\cdots,n)$ 为矩阵 A 的奇异值。

7.矩阵范数

函数 $\|\cdot\|:\mathbb{R}^{m\times n}\to\mathbb{R}$ 称为一个矩阵范数，如果对任意 $m\times n$ 矩阵 A,B 及实数 a 满足：

①非负性：$\|A\|\geq 0$，当且仅当 $A=O$ 时 $\|A\|=0$　　**②齐次性**：$\|aA\|=|a|\|A\|$

③三角不等式：$\|A+B\|\leq\|A\|+\|B\|$　　**④相容性**：$\|AB\|\leq\|A\|\|B\|$

满足①~③的范数称为矩阵上的向量范数(Vector Norm on Matrix)。

Frobenius 范数：假设矩阵 $A\in\mathbb{R}^{m\times n}$，则称 $\|A\|_F=\sqrt{\mathrm{tr}(AA^T)}=\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|A_{ij}|^2}$ 为矩阵 A 的 F 范数，也称 l_2 范数。

1-范数(L_1 范数)：$\|x\|_1=|x_1|+|x_2|+\cdots+|x_n|$

2-范数(L_2 范数)：$\|x\|_2=\sqrt{|x_1|^2+|x_2|^2\cdots+|x_n|^2}$

∞-范数(L_∞ 范数)：$\|x\|_\infty=\max\{|x_1|,|x_2|,\cdots,|x_n|\}$

8.奇异值分解(SVD 矩阵分解)

设矩阵 $A_{m\times n}$ 的秩 $r>0$，则存在 m 阶正交矩阵 U 和 n 阶正交矩阵 V 使得 $A=U\begin{pmatrix}\Sigma & O\\ O & O\end{pmatrix}V^T$，其中 $\Sigma=\mathrm{diag}(\sigma_1,\sigma_2,\cdots,\sigma_r)$，为矩阵 A 的全部非零奇异值构成的对角矩阵，称此为矩阵 A 的奇异值分解。

如,求 $A=\begin{pmatrix}1&0&1\\0&1&1\\0&0&0\end{pmatrix}$ 的 SVD 分解：

记 $B=A^TA=\begin{pmatrix}1&0&1\\0&1&1\\1&1&2\end{pmatrix}$，特征方程 $0=|\lambda E-B|=\begin{vmatrix}\lambda-1&0&-1\\0&\lambda-1&-1\\-1&-1&\lambda-2\end{vmatrix}=\lambda(\lambda-3)(\lambda-1)$，得 B 特征值为 $3,1,0$，则 A 的奇异值为 $\sqrt{3},1,0$。B 的特征值对应的特征向量解出来为 $\begin{pmatrix}1\\1\\2\end{pmatrix},\begin{pmatrix}1\\-1\\0\end{pmatrix},\begin{pmatrix}1\\1\\-1\end{pmatrix}$。

$\mathrm{r}(A)=2,\Sigma=\begin{pmatrix}\sqrt{3}&0\\0&1\end{pmatrix}$，$V$ 为归一化后的特征向量拼接成的矩阵 $V=\begin{pmatrix}\frac{1}{\sqrt{6}}&\frac{1}{\sqrt{2}}&\frac{1}{\sqrt{3}}\\\frac{1}{\sqrt{6}}&-\frac{1}{\sqrt{2}}&\frac{1}{\sqrt{3}}\\\frac{2}{\sqrt{6}}&0&-\frac{1}{\sqrt{3}}\end{pmatrix}$，

$U_1=AV_1\Sigma^{-1}=\begin{pmatrix}\frac{1}{\sqrt{2}}&\frac{1}{\sqrt{2}}\\\frac{1}{\sqrt{2}}&-\frac{1}{\sqrt{2}}\\0&0\end{pmatrix},U_2=\begin{pmatrix}0\\0\\1\end{pmatrix},U=(U_1:U_2)=\begin{pmatrix}\frac{1}{\sqrt{2}}&\frac{1}{\sqrt{2}}&0\\-\frac{1}{\sqrt{2}}&-\frac{1}{\sqrt{2}}&0\\0&0&1\end{pmatrix}$

二、概率统计

1.三个公式

①条件概率 $P(B|A)=\dfrac{P(AB)}{P(A)}$　　$P(A_1\cdots A_n)=P(A_1)P(A_2|A_1)P(A_3|A_1A_2)\cdots P(A_n|A_1\cdots A_{n-1})$

②全概率公式 $P(B)=\sum_{i=1}^{n}P(A_i)P(B|A_i)$

③贝叶斯公式 $P(A|B)=\frac{P(A)P(B|A)}{P(B)}$　　$P(A_j|B)=\frac{P(A_j)P(B|A_j)}{\sum_{i=1}^{n}P(A_i)P(B|A_i)}$

2.信息论

①基本思想：越不可能发生的事情发生了，其包含的信息量就越大。

②熵：表述随机变量不确定度的度量。熵越大，意味着发生的可能性越大。

$$H(X)=-\sum_{x\in X}p(x)\log p(x)$$

所有样本等几率出现的情况下，熵达到最大值（所有可能的事件等概率时不确定性最高）

对于样本等几率分布而言，样本数越大，熵值越大（可能的事件越多，不确定性越高）

KL 散度 (相对熵)：

$$KL(P\|Q)=\int_{-\infty}^{+\infty}p(x)\log\frac{p(x)}{q(x)}\mathrm{d}x=\int_{-\infty}^{+\infty}p(x)\log p(x)\mathrm{d}x-\int_{-\infty}^{+\infty}p(x)\log q(x)\mathrm{d}x=-H(P)+H(P,Q)$$

KL 散度可认为表示使用基于 Q 的编码对来自 P 的编码所需要的额外字节数。

KL 散度不满足对称性，即 $KL(P\|Q)\neq KL(Q\|P)$

交叉熵：

$$H(P,Q)=-\int_{-\infty}^{+\infty}p(x)\log q(x)\mathrm{d}x$$

交叉熵可认为表示使用基于 Q 的编码对来自 P 的编码所需要的字节数。

条件熵：

$$H(Y|X)=\sum_{x\in X}p(x)H(Y|X=x)=-\sum_{x\in X}p(x)\sum_{y\in Y}p(y|x)\log p(y|x)=-\sum_{x\in X}\sum_{y\in Y}p(x,y)\log p(y|x)$$

三、优化理论

1.设 $\nabla^2 f(\vec{x})$ 是 $f(\vec{x})$ 在 $\vec{x}$ 处的二阶导数矩阵或者 Hessian 矩阵，则

(1) f 是凸集 $\vec{S}$ 上的凸函数 $\Leftrightarrow \forall\vec{x}\in\vec{S},\nabla^2 f(\vec{x})$ 半正定

(2) $\forall\vec{x}\in\vec{S},\nabla^2 f(\vec{x})$ 正定 $\Rightarrow$ f 是凸集 $\vec{S}$ 上的严格凸函数

2.典型的凸函数

(1)log-sum-exp 函数 $f(\vec{x})=\log(\sum_{i=1}^{n}e^{x_i})$ （常称为 softmax 函数）

(2)线性函数、指数函数、负熵、范数等

3.凸优化问题

凸优化问题(Convex optimization problem)要求目标函数为凸函数，而且定义域为凸集，即要求目标函数 $f_0(x)$ 和约束函数 $f_i(x)(i=1,\ldots,m)$ 均为凸函数

对凸优化问题，局部最优就是全局最优。

4.多元函数的极大小值问题

(1)无约束的优化问题

驻点条件： $\nabla f(\vec{x})=\vec{0}$

考虑 Hessian 矩阵 $H(\vec{x})=\begin{bmatrix}\frac{\partial^2 f}{\partial x_1\partial x_1} & \frac{\partial^2 f}{\partial x_1\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\partial x_d}\\ \frac{\partial^2 f}{\partial x_2\partial x_1} & \frac{\partial^2 f}{\partial x_2\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_2\partial x_d}\\ \vdots & \vdots & & \vdots\\ \frac{\partial^2 f}{\partial x_d\partial x_1} & \frac{\partial^2 f}{\partial x_d\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_d\partial x_d}\end{bmatrix}$

如果 $H(x^*)$ 是正定的，则 x^* 是极小值点；

如果 $H(x^*)$ 是负定的，则 x^* 是极大值点；

如果 $H(x^*)$ 是不定的，则 x^* 是鞍点，即在某个方向上有极小值，在另一方向上有极大值。

如，求 $f(x,y)=3x^2+2y^3-2xy$ 的驻点。

解：$\nabla f=\begin{bmatrix}6x-2y\\ 6y^2-2x\end{bmatrix}=\vec{0}\Rightarrow\begin{cases}x^*=0\\ y^*=0\end{cases}OR\begin{cases}x^*=\frac{1}{27}\\ y^*=\frac{1}{9}\end{cases}$。Hessian 矩阵为 $H(x,y)=\begin{bmatrix}6 & -2\\ -2 & 12y\end{bmatrix}$，在 $x^*=y^*=0$ 处是不定的，因此为鞍点；在 $x^*=\frac{1}{27},y^*=\frac{1}{9}$ 处是正定的，因此 $(\frac{1}{27},\frac{1}{9})$ 是一个极小值点，极小值为 –0.0014。

(2)有约束的优化问题

设优化目标 $f(\vec{x})$，约束条件 $g(\vec{x})$

构造拉格朗日函数 $L(\vec{x},\lambda)=f(\vec{x})+\lambda g(\vec{x})$，有 $\begin{cases}\nabla_{\vec{x}}L(\vec{x},\vec{\lambda})=0\\ \nabla_{\vec{\lambda}}L(\vec{x},\vec{\lambda})=0\end{cases}$

(3)不等式约束的优化问题

(4)拉格朗日对偶性

5.优化方法

(1)梯度下降法 Gradient Descent/最速下降法 Steepest Descent

(2)牛顿法 Newton Method

四、图论

1.谱聚类算法

输入：样本集 $G=(x_1,x_2,\ldots,x_n)$，相似矩阵的生成方式，降维后的维度 k_1，聚类方法，聚类后的维度 k_2

输出：簇划分 $C=(c_1,c_2,\ldots,c_{k_2})$

过程：①根据输入发相似矩阵的生成方式构建样本集的相似矩阵 S

②根据 S 构建邻接矩阵 W，构建度矩阵 A

③计算拉普拉斯矩阵 $L=D-A$

④构建标准化的拉普拉斯矩阵 $D^{-\frac{1}{2}}LD^{\frac{1}{2}}$

⑤计算 $D^{-\frac{1}{2}}LD^{\frac{1}{2}}$ 最小的 k_1 个特征值对应的特征向量 $\vec{f}$

⑥将各自对应的 $\vec{f}$ 组成的矩阵标准化，最终组成 $n\times k_1$ 维的特征矩阵 F

⑦对 F 中的每一行作为 k_1 维的样本，共 n 个，用输入的聚类算法进行聚类，聚类维数为 k_2

⑧得到簇划分 $C=(c_1,c_2,\ldots,c_{k_2})$

【例】以下图为例简单介绍谱聚类实现。

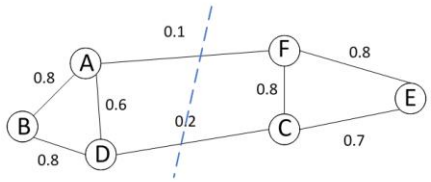


图 2-35 图G的网络结构

相似矩阵$\begin{bmatrix} 0 & 0.8 & 0.6 & 0 & 0.1 & 0 \\ 0.8 & 0 & 0.8 & 0 & 0 & 0 \\ 0.6 & 0.8 & 0 & 0.2 & 0 & 0 \\ 0 & 0 & 0.2 & 0 & 0.8 & 0.7 \\ 0.1 & 0 & 0 & 0.8 & 0 & 0.8 \\ 0 & 0 & 0 & 0.7 & 0.8 & 0 \end{bmatrix}$，计算度矩阵$\boldsymbol{D} = \begin{bmatrix} 1.5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.5 \end{bmatrix}$

求G的拉普拉斯矩阵为：

$$\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A} = \begin{bmatrix} 1.5 & -0.8 & -0.6 & 0 & -0.1 & 0 \\ -0.8 & 1.6 & -0.8 & 0 & 0 & 0 \\ -0.6 & -0.8 & 1.6 & -0.2 & 0 & 0 \\ 0 & 0 & -0.2 & 1.7 & -0.8 & -0.7 \\ -0.1 & 0 & 0 & -0.8 & 1.7 & -0.8 \\ 0 & 0 & 0 & -0.7 & -0.8 & 1.5 \end{bmatrix}$$

对其进行标准化：

$$\hat{\boldsymbol{L}} = \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{L}\boldsymbol{D}^{\frac{1}{2}} = \begin{bmatrix} 1 & -0.52 & -0.39 & 0 & -0.06 & 0 \\ -0.52 & 1 & -0.5 & 0 & 0 & 0 \\ -0.39 & -0.5 & 1 & -0.12 & 0 & 0 \\ 0 & 0 & -0.12 & 1 & -0.47 & -0.44 \\ -0.06 & 0 & 0 & -0.47 & 1 & -0.5 \\ 0 & 0 & 0 & -0.44 & -0.5 & 1 \end{bmatrix}$$

对标准化的矩阵求特征值和特征向量，求得特征向量矩阵：

$$\begin{bmatrix} 0.4026 & -0.3963 & -0.5191 & -0.3751 & -0.3452 & -0.3892 \\ 0.4163 & -0.4434 & -0.0973 & 0.2464 & 0.0301 & 0.7476 \\ 0.4146 & -0.3729 & 0.6023 & 0.1613 & 0.3276 & -0.4399 \\ 0.4142 & 0.3849 & 0.4810 & -0.4317 & -0.4622 & 0.2211 \\ 0.4127 & 0.4193 & -0.2943 & -0.2813 & 0.6972 & 0.0452 \\ 0.3883 & 0.4282 & -0.2009 & 0.7120 & -0.2711 & -0.2122 \end{bmatrix}$$

同时求得对应的特征值为：-0.0008，0.1148，1.3210，1.4643，1.5357，1.5650。

之后可以选择$K-Means$等聚类方法对求得的特征向量进行聚类得到簇划分，后续步骤限于篇幅过程暂不展开。

若将得到的结果分为两类，则左边三个点为一类，右边三个点为一类，这也很符合直观的观察结果，左边的三个点之间有较为紧密的联系，右边的三个点间也有较为紧密的联系。

代码部分

一、基础部分

1.NumPy

NumPy 是 Numerical Python 的缩写。numerical ： relating to numbers; expressed in numbers

初识 numpy——列表/元组转为数组

```
import numpy as np

list_demo = [1, 2, 3, 4]
a = np.array(list_demo)
print(a)

tuple_demo = ('1', '2', '3', '4')
b = np.asarray(tuple_demo)
print(b)

list_tuple = [('胡桃', '可莉', '纳西妲'), ('7.15', '7.27', '10.27')]
c = np.asarray(list_tuple)
print(c)
```

输出

```
[1 2 3 4]
['1' '2' '3' '4']
[['胡桃' '可莉' '纳西妲']
 ['7.15' '7.27' '10.27']]
```

第三个的输出：将列表转换为 NumPy 数组时，NumPy 会将列表中的元组视为一个整体，并将该整体作为数组的一个元素。

初识 numpy——创建数值数组

```
array1 = np.arange(0, 10, 2, int)
print(array1)

array2 = np.arange(0, 1, 0.2, float)
print(array2)

array3 = np.linspace(0, 1, num=5, endpoint=False, retstep=True)
print(array3)
```

输出

```
[0 2 4 6 8]
[0.  0.2 0.4 0.6 0.8]
(array([0. , 0.2, 0.4, 0.6, 0.8]), 0.2)
```

函数 **arange** 用于创建一个等差数列的一维数组，参数包括起始值、结束值（不包含在数组中）、步长和数据类型，其中起始值默认 0，步长默认为 1，数据类型参数可省略。

函数 **linspace** 用于在一定范围内生成等间隔的数值序列，其中参数 num 表示生成数组的元素个数，endpoint 表示是否包含结束点，retstep 表示是否返回步长。

其他方法：

ones(shape,dtype)：指定数组的形状，生成一个包含全 1 元素的数组。数据类型为 dtype，可省略。

eye(N)：指定方阵的大小 N，生成一个单位矩阵。

zeros(shape,dtype)：指定数组的形状，生成一个包含全 0 元素的数组。数据类型为 dtype，可省略。

注：dtype 可省略，shape 可以只有一个，默认为行数。

比如

```
print(np.eye(2))
print(np.ones((3, 2)))
print(np.zeros((3, 2)))
```

输出

```
[[1. 0.]
 [0. 1.]]
[[1. 1.]
 [1. 1.]
 [1. 1.]]
[[0. 0.]
 [0. 0.]
 [0. 0.]]
```

注意到这里的 ones 与 zeros 都是用的((a,b))，不要只打一个括号。

事实上有 **eye(N, M=None, k=0, dtype=float)**，这个比较复杂，单独说。它创建一个大小为(N, M)的二维数组，其中“对角线”上的元素为 1，其他元素为 0。偏移量 k 表示“对角线”向右的偏移量。比如：

```
e = np.eye(5, 6, -1, int)
print(e)
```

输出

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

-1 表示向左偏移 1。

Ndarray 对象

Ndarray 对象是 NumPy 库中的多维数组对象，集合了 N 个类型相同的数据，shape 与 dtype 分别反应了数组维度与数据类型情况。创建 Ndarray 可以用 array()方法，上文已经使用过了。原来是嵌套格式的序列会被转换成多维数组。

对 Ndarray 对象使用**.ndim** 与**.dtype** 可以获取数组维度数与数据类型，用**.shape** 与**.size** 可以获取大小(行,列)与元素数量，用**. itemsize** 可以获取元素的字节数。

从外向里依次“剥皮”，也就是去掉[]，就分别是数组的第 1,2,3,.....维度，对应的是 shape[0],shape[1],shape[2]…。而 shape[n]的值的计算方法是：去掉外面的中括号之后，内部的最大的中括号有几个。

比如下文中去掉最外面的[]后，得到的单位体的个数表示第 0 个维度(**axis=0**)的大小，剩下的是[[1, 2, 3, 4],[1, 2, 3, 4],[1, 2, 3, 4]]，只有一个括号，所以 shape[0]为 1。

接着去掉[]后，剩下[1, 2, 3, 4],[1, 2, 3, 4],[1, 2, 3, 4]，有三个括号，所以 shape[1]为 3。

然后内部元素是 4 个，shape[2]为 4。

```
list_demo = [[[1, 2, 3, 4],[1, 2, 3, 4],[1, 2, 3, 4]]]
a = np.array(list_demo)
print(a.ndim)
print(a.dtype)
print(a.shape)
```

输出

```
3
int32
```

(1, 3, 4)

上文提及了参数 axis，该参数在 sum 求和里比较重要：

```
print(a.sum(axis=0))
print(a.sum(axis=1))
print(a.sum(axis=2))
```

输出

```
[[1 2 3 4]
 [1 2 3 4]
 [1 2 3 4]]
[[ 3  6  9 12]]
[[10 10 10]]
```

其中 axis=0 说明去掉一个[]，这时候 shape[0]=1，此时就是一个数组，sum 运算后还是该数组。axis=1 说明去掉两层[]，这时候 shape[1]=3，此时就是三个数组[1,2,3,4]，将这三个数组相加得到 sum。axis=2 说明去掉三层[]，这时候 shape[2]=4，将 1,2,3,4 相加就是 10。

现在应该大致明白了计算方法，下面来谈谈这个输出里面的[]个数如何确定。因为 shape[0]=1 的时候不方便观察和(因为 axis=0 时没有相加过程)，所以这里取 shape[0]=2 的一个数组，比如 shape=(2,4,3)。比如

```
arr = [[[5, 0, 3], [3, 7, 3], [5, 2, 4], [7, 6, 8]], [[8, 1, 6], [7, 7, 8], [1, 5, 8], [4, 3, 0]]]
arr = np.asarray(arr)
print(arr, end='\n\n')
print(arr.sum(axis=0), end='\n\n')
print(arr.sum(axis=1), end='\n\n')
print(arr.sum(axis=2), end='\n\n')
```

输出见右边文本框

```
输出
[[[5 0 3]
  [3 7 3]
  [5 2 4]
  [7 6 8]]

 [[8 1 6]
  [7 7 8]
  [1 5 8]
  [4 3 0]]]

[[13  1  9]
 [10 14 11]
 [ 6  7 12]
 [11  9  8]]

[[20 15 18]
 [20 16 22]]

[[ 8 13 11 21]
 [15 22 14  7]]
```

axis=0 时，去掉最外层的[]，此时两个[[…]]的数组相加。
axis=1 时，去掉中间的[]，此时[[[…]]]变为[[…]](但这与 axis=0 的时候的[[…]]是不一样的，因为去掉的那个[]不一样)。
axis-2 时，去掉最内层的[]，此时是数字元素之间的相加了。

显式转换

```
arr = [[0.0, 1.0, 1.1], [-1.5, 1.5, -1.6]]
arr = np.array(arr)
print(arr)
print(arr.astype(np.int32))
```

输出

```
[[ 0.   1.   1.1]
 [-1.5  1.5 -1.6]]
[[ 0  1  1]
 [-1  1 -1]]
```

这里类型转化的时候的小数点是被截断的。

数组的操作

NumPy 数组与普通的列表的操作差不多，这里仅说明一些比较重要的。

使用切片修改原数据

```
arr = [1, 2, 3, 4, 5]
arr = np.array(arr)
arr_slice = arr[2:4]  # 截取的是元素 3,4
arr_slice[1] = 44  # 把截取的元素中的第二个(这里是 4)改为 44
print(arr)
```

输出[1 2 3 44 5]

如果想要 arr_slice 里的元素全部更改，可以用 arr_slice[:]=44，不要忘记:了。

多维下标索引

和 C 里面的基本一样，比如

```
arr = [[1, 2, 3], [4, 5, 6]]
arr = np.array(arr)
print(arr[0], end='\t')
print(arr[0][1], end='\t')
print(arr[0, 1])
```

输出[1 2 3]2 2

但是，如果需要的是某一部分的元素，难度比较大，需要索引与切片结合。比如

```
print(arr[:1, 2:])
```

输出[[3]]

这里输出的是行为:1，列为 2:的元素。切片:1 表示结束位置为 1 且不包括 1，2:表示起始位置为 2 且包括 2。又如

```
print(arr[1:, 1:2])
print(arr[:, 1:])
```

输出

```
[[5]]
[[2 3]
 [5 6]]
```

亦可对切片赋值：

```
arr[:, 1:] = 0
print(arr)
```

输出

```
[[1 0 0]
[4 0 0]]
```

转置

```
arr = [[1, 2, 3], [4, 5, 6]]
arr = np.array(arr)
print(arr.T)
print(arr.transpose())
```

输出均为

```
[[1 4]
 [2 5]
 [3 6]]
```

```
arr = np.arange(32).reshape((2, 4, 4))
print(arr)
print(arr.shape)
print(arr.transpose(1, 0, 2))
print(arr.transpose(1, 0, 2).shape)
```

输出如右文本框

```
输出
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]
  [12 13 14 15]]

 [[16 17 18 19]
  [20 21 22 23]
  [24 25 26 27]
  [28 29 30 31]]]
(2, 4, 4)
[[[ 0  1  2  3]
  [16 17 18 19]]

 [[ 4  5  6  7]
  [20 21 22 23]]

 [[ 8  9 10 11]
  [24 25 26 27]]

 [[12 13 14 15]
  [28 29 30 31]]]
(4, 2, 4)
```

这里(1,0,2)表示第一个维度(0)与第二个维度(1)交换，第三个维度(2)不变。也就是 shape 从(2,4,4)变为(4,2,4)。之前 axis=0 的维度里 0123 后面的是 16171819，所以现在 axis=1 的维度里 0123 后面的是 16171819，以此类推。

生成随机数组

```
print(np.random.randn(3,4))
print(np.random.choice([10,20,30]))
print(np.random.beta(1,5,10))
```

输出

```
[[ 1.62816762  1.47448347  1.00854331  1.47853539]
 [-0.19053414 -1.06565901  0.33980349 -1.31290175]
 [-0.16290617 -0.5937015   0.82524027  1.56946873]]
10
[0.01761509 0.21731346 0.04291937 0.12523884 0.12166361 0.14969903
 0.03939118 0.17141757 0.02659418 0.04199945]
```

np.random.randn 函数生成的是服从标准正态分布 $N \sim (0,1)$ 的随机数。

np.random.choice 从给的的数组里随机选一个元素。

np.random.beta 是一个使用 beta 分布生成随机数的函数。前两个参数是 beta 分布的形状参数，第三个参数表示生成的随机数的数量。

Beta 分布是一种连续型概率密度分布，表示为 $x \sim Beta(a,b)$，由两个参数 a,b 决定，称为形状参数。由于其定义域为(0,1)，一般被用于建模伯努利试验事件成功的概率的概率分布：为了测试系统的成功概率，我们做 n 次试验，统计成功的次数 k，于是很直观地就可以计算出。然而由于系统成功的概率是未知的，这个公式计算出的只是系统成功概率的最佳估计。也就是说实际上也可能为其它的值，只是为其它的值的概率较小。因此我们并不能完全确定硬币出现正面的概率就是该值，所以也是一个随机变量，它符合 Beta 分布，其取值范围为 0 到 1。参考 https://zhuanlan.zhihu.com/p/69606875，https://zhuanlan.zhihu.com/p/149964631。

```
print(np.random.rand(2,4))
print(np.random.rand())
print(np.random.randint(1,10,3))
```

输出

```
[[0.46653209 0.35644468 0.83639798 0.27475056]
 [0.68402447 0.93758497 0.44335841 0.22709871]]
0.6939882918736527
[5 5 1]
```

np.random.rand 生成 0~1 的随机数

np.random.randint(1,10,3)生成 1~10 之间的长度为 3 的随机整数数组

np.array()和 np.asarray()的区别

```
import numpy as np

arr1 = np.ones((3, 3))
arr2 = np.array(arr1)
arr3 = np.asarray(arr1)
arr1[1] = 2
print('arr1:\n', arr1)
print('arr2:\n', arr2)
print('arr3:\n', arr3)
```

输出

```
arr1:
 [[1. 1. 1.]
 [2. 2. 2.]
 [1. 1. 1.]]
arr2:
```

```
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
arr3:
 [[1. 1. 1.]
 [2. 2. 2.]
 [1. 1. 1.]]
```

发现 arr3 被 arr1 改了，这是因为：
np.array(默认情况下)将会 copy 该对象，而 np.asarray 除非必要，否则不会 copy 该对象。(必要的意思是数据源是 ndarray 类型时，不会 copy)。也就是说 array 和 asarray 都可以将结构数据转化为 ndarray，但是主要区别就是当数据源是 ndarray 时，array 仍然会 copy 出一个副本，占用新的内存，但 asarray 不会。

矩阵的乘法——星乘*与点乘 dot：

1.星乘*

①同型矩阵(哈达玛积)。对应位置的元素相乘 $c_{ij}=a_{ij}\times b_{ij}$：

如 $\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} aA & bB \\ cC & dD \end{pmatrix}$

②不同型，但两个矩阵行数相等，其中一个矩阵列数为 1。单列矩阵的列与另一个矩阵的列分别相乘：

如 $\begin{pmatrix} a \\ b \end{pmatrix} * \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} aA & aB \\ bC & bD \end{pmatrix}$

2. 点乘 dot

就是线代里的矩阵乘法

比如：

```python
import numpy as np

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
c = np.array([[1], [2]])
print(a * b, '\n')
print(c * b, '\n')
print(b * c, '\n')
print(np.dot(a, b))
```

输出

```
[[ 5 12]
 [21 32]]

[[ 5  6]
 [14 16]]

[[ 5  6]
 [14 16]]

[[19 22]
 [43 50]]
```

矩阵的逆矩阵：

与线代的定义一样，$AA^{-1}=E$，比如：

```python
import numpy as np

a = np.array([[1, 2], [3, 4]])
a_inv = np.linalg.inv(a)
print(a_inv)
print(np.dot(a, a_inv))
```

输出

```
[[-2.    1. ]
 [ 1.5 -0.5]]
[[1.0000000e+00 0.0000000e+00]
 [8.8817842e-16 1.0000000e+00]]
```

inv 是 inverse 的缩写：相反的，倒转的；颠倒的，逆的

linalg 是 numpy 库中的线性代数模块，可以用于进行矩阵和向量的运算，linalg 是 linear algebra(线性代数)的缩写。

这里有烦人的浮点数，可以这样修改代码：

```
a_dot_a_inv = np.dot(a, a_inv)
a_dot_a_inv = np.round(a_dot_a_inv).astype(int)  # 将 a_dot_a_inv 中的元素四舍五入为整数
print(a_dot_a_inv)
```

输出

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

2.Pandas

Series 数据结构

创建 Series 数组

```
import pandas as pd

obj = pd.Series([2, 4, 3, 1])
print(obj)
print(obj.values)
print(obj.index)
```

输出

```
0    2
1    4
2    3
3    1
dtype: int64
[2 4 3 1]
RangeIndex(start=0, stop=4, step=1)
```

series 类似一维数组，由一组数据与对应的数据标签组成。

索引在左，值在右，并注明 dtype。还可以在创建时指定索引：

```
obj1 = pd.Series([2, 4, 3, 1], index=['b','D','C','a'])
print(obj1)
print(obj1.values)
print(obj1.index)
```

输出

```
b    2
D    4
C    3
a    1
dtype: int64
[2 4 3 1]
Index(['b', 'D', 'C', 'a'], dtype='object')
```

索引

```
print(obj1[1])
print(obj1['D'])
```

输出均为 4

```
print(obj1[[1, 2]])
```

输出

```
D    4
```

```
C    3
dtype: int64
```

```
obj1.index = ["a", 'bb', 'cc', 'd']
print(obj1)
```

输出

```
a     2
bb    4
cc    3
d     1
dtype: int64
```

用赋值的方式可以修改 index

注意引用索引时，利用标签 index 的切片运算的末端是包含在内的，这与普通的切片运算不同。

重新索引

```
data = pd.Series([7, 1, 5, 2, 3], index=['H', 'u', 'T', 'a', 'o'])
data = data.reindex(['H', 'u', 't', 'a', 'o', 'T'], fill_value=0)
print(data)
```

输出

```
H    7
u    1
t    0
a    2
o    3
T    5
dtype: int64
```

不指定 fill_value 就默认为 NAN。注意这个 reindex 可不是随意指定的，只要是与之前的 index 不同就会视为没有值，也就是 NAN。

丢弃

drop()可以丢弃指定行/列

```
data = pd.Series([7, 1, 5, 2, 3], index=['H', 'u', 'T', 'a', 'o'])
data = data.drop('a')
print(data)
```

输出

```
H    7
u    1
T    5
o    3
dtype: int64
```

DataFrame 数据结构

创建 DataFrame

DataFrame 是表格类型，有多个列，每个列都能设置不同的类型(字符串、数值、布尔值等)，需要保证它们长度相等。

```
import pandas as pd

data = {'name': ["Hu Tao", "Klee", "Keqing"],
        'bir': [7.15, 7.27, 11.20],
        'num': [1, 2, 3]}  # 创建字典
```

```
frame = pd.DataFrame(data)  # 使用字典创建数据表
print(frame)
```

输出

```
     name    bir  num
0  Hu Tao   7.15    1
1    Klee   7.27    2
2  Keqing  11.20    3
```

也可以指定列顺序：

```
frame = pd.DataFrame(data, columns=['num', 'bir', 'name'])
print(frame)
```

输出

```
   num    bir    name
0    1   7.15  Hu Tao
1    2   7.27    Klee
2    3  11.20  Keqing
```

获取 DataFrame 的某一列作为一个 Series：

```
print(frame['name'])
print(frame.name)
```

均输出

```
0    Hu Tao
1      Klee
2    Keqing
Name: name, dtype: object
```

丢弃

```
import pandas as pd

data = {'name': ["Hu Tao", "Klee", "Keqing"],
        'bir': [7.15, 7.27, 11.20],
        'num': [1, 2, 3]}
frame = pd.DataFrame(data)
frame = frame.drop(1)
frame = frame.drop('name', axis=1)
print(frame)
```

输出

```
     bir  num
0   7.15    1
2  11.20    3
```

先删除 index=1 的行(第二行)[因为这里没有指定 index，所以按默认的来]，再删除 name 列(要注明 axis=1，指定在列上操作。默认值 axis=0 表示行)

指定 index 要按如下方法删除(再用 1 就会报错了 KeyError: '[1] not found in axis')：

```
frame = pd.DataFrame(data,index=['a','b','c'])
frame = frame.drop('b')
```

索引

```
data = pd.DataFrame(np.arange(16).reshape((4,4)),
                    index=['A', 'B ', 'C', 'D'],
                    columns=['a', 'b', 'c', 'd'])
print(data)
print(data.loc['C': 'D', 'b'])
```

输出

```
   a   b   c   d
```

```
A    0   1   2   3
B    4   5   6   7
C    8   9  10  11
D   12  13  14  15
C     9
D    13
Name: b, dtype: int32
```

这里如果截取的是某一列或者某一行，就会显示 Name 与 dtype，否则就像全部输出一样只有 index,columns 与元素。

算术运算

如果索引相同则对对应内容加减，若有不同的索引，则该索引对应的值为 NAN。

以 Series 为例：

```
obj1 = pd.Series([1, 2, 3, 4], index=['a','b','c','d'])
obj2 = pd.Series([12, 34, 56, 78], index=['a','B','c','d'])
print(obj1+obj2)
```

输出

```
B     NaN
a    13.0
b     NaN
c    59.0
d    82.0
dtype: float64
```

输出是浮点数因为在 Panda 中，当进行数据计算时，如果有一个操作数是浮点数，则结果的数据类型会自动转换为浮点数。NaN（Not a Number）值的数据类型是 float。

以 DataFrame 为例：

```
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)),
                   columns=list('abc'),
                   index=list('ABC'))
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)),
                   columns=list('abc'),
                   index=list('ABCD'))
print(df1+df2)
```

输出

```
     a     b     c
A   0.0   2.0   4.0
B   6.0   8.0  10.0
C  12.0  14.0  16.0
D   NaN   NaN   NaN
```

以 DataFrame 与 Series 运算为例：

```
series = df2.loc['A']
print(df2-series)
```

输出

```
     a    b    c
A  0.0  0.0  0.0
B  3.0  3.0  3.0
C  6.0  6.0  6.0
D  9.0  9.0  9.0
```

这里是将 Series 的列匹配到 DataFrame 的每一列，若有一方索引不对应，则值为 NAN。其中 df2 与 series 为：

```
     a     b     c
A  0.0   1.0   2.0
B  3.0   4.0   5.0
C  6.0   7.0   8.0
D  9.0  10.0  11.0
```

与

```
a    0.0
b    1.0
c    2.0
Name: A, dtype: float64
```

用 0,1,2 对应去减 a,b,c 列。

3.SciPy

保存与读取矩阵文件

```
from scipy import io
import numpy as np
arr = np.array([1,2,3,4,5,6])
io.savemat('scipy_test.mat',{'arr1':arr})  # 将数据保存到.mat 文件，其中'arr1'是在.mat 文件中存储的数组的键名
loadArr = io.loadmat('scipy_test.mat')  # 从.mat 文件中加载数据
print(loadArr['arr1'])
```

输出[[1 2 3 4 5 6]]

在当前文件夹保存 scipy_test.mat 文件，使用 MATLAB 打开如下：

统计功能

```
import scipy.stats as stats
```

均匀分布

```
x = stats.uniform.rvs(size=10)
```

正态分布

```
x = stats.norm.rvs(size=10)
```

贝塔分布

```
x = stats.beta.rvs(size=10, a=2, b=3)
```

泊松分布

```
x = stats.poisson.rvs(0.8, loc=0, size=10)  # 0.8 是泊松分布的参数 λ，表示平均发生率或速率。loc=0 表示事件发生的起始值，默认为 0。
```

均值与标准差计算

```
x = np.array([0,1,2,3,4])
print(stats.norm.fit(x))
```

输出

```
(2.0, 1.4142135623730951)
```

偏度计算

```
x = np.array([0.1, 0.2, 0.11, 0.22, 0.3, 0.5, -0.2, -0.4, -0.715, 0.727])
print(stats.skewtest(x))
```

输出

```
SkewtestResult(statistic=-0.8048934446906357, pvalue=0.42088117154954274)
```

偏度描述的是概率分布的偏离(非对称)程度。

$$Skew(X)=E(\frac{X-\mu}{\sigma})=\frac{k_3}{\sigma_3}=\frac{k_3}{k_2^{\frac{3}{2}}}=\frac{\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^3}{[\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2]^{\frac{3}{2}}}$$。

$Skew(X)<0$：左偏。$Skew(X)=0$：正态分布。$Skew(X)>0$：右偏

skewtest 函数最少需要 8 个参数。

statistic：float。The computed z-score for this test.

pvalue：float。The p-value for the hypothesis test.

statistic 是偏度检验的统计量，它用来衡量数据的偏斜程度。

pvalue 是偏度检验的 p 值，它用来判断统计量的显著性。如果 p 值小于显著性水平（通常为 0.05），则拒绝原假设，即认为数据是有偏的；如果 p 值大于显著性水平，则接受原假设，即认为数据是无偏的。这里指的是数据集服从正态分布的概率(0~1)。

峰度计算

```
x = np.array([0.1, 0.2, 0.11, 0.22, 0.3, 0.5, -0.2, -0.4, -0.715, 0.727])
print(stats.kurtosis(x))
```

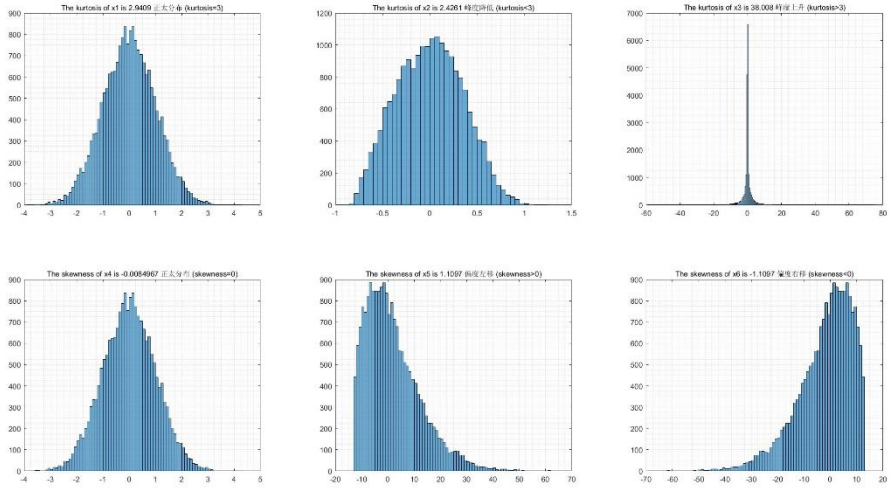
输出-0.4344773135788995

峰度描述的是概率分布曲线的陡峭程度。

$$Kurt(X)=E(\frac{X-\mu}{\sigma})^4=\frac{E(X-\mu)^4}{E(X-\mu)^2}=\frac{\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^4}{[\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2]^2}$$

$Kurt(X)<3$：平峰，薄尾。$Kurt(X)=3$：正态分布。$Kurt(X)>3$：尖峰，厚尾。

偏度与峰度的样子如下图：（来源 https://zhuanlan.zhihu.com/p/346534221）



正态分布程度检验

```
x = np.array([0.1, 0.2, 0.11, 0.22, 0.3, 0.5, -0.2, -0.4, -0.715, 0.727, 0.1, 0.2,
0.11, 0.22, 0.3, 0.5, -0.2, -0.4, -0.715, 0.727])
print(stats.normaltest(x))
```

输出 NormaltestResult(statistic=0.9975897425378244, pvalue=0.6072620478568534)

该函数需要至少 20 个参数，否则给出 warning：

UserWarning: kurtosistest only valid for n>=20 ... continuing anyway, n=10

warnings.warn("kurtosistest only valid for n>=20 ... continuing "

但还是能给出结果

计算某一百分比处的数值

```
x = np.array([0.1, 0.2, 0.11, 0.22, 0.3, 0.5, -0.2, -0.4, -0.715, 0.727])
print(stats.scoreatpercentile(x, 50))
```

输出 0.155

这里计算的是中位数。

SciPy 应用

```
import numpy as np
from scipy import stats

arr = np.array(
    [[0, 2], [2.5, 4], [5, 6], [7.5, 9], [10, 13], [12.5, 16], [15, 19], [17.5, 23],
[20, 27], [22.5, 31], [25, 35],
     [27.5, 40], [30, 53], [32.5, 68], [35, 90], [37.5, 110], [40, 130], [42.5,
148], [45, 165], [47.5, 182], [50, 195],
     [52.5, 208], [55, 217], [57.5, 226], [60, 334], [62.5, 342], [65, 349], [67.5,
500], [70, 511], [72.5, 300],
     [75, 200], [77.5, 80], [80, 20], [82.5, 50], [85, 6], [90, 3]])
score, num = arr[:, 0], arr[:, 1]  # 所有行第一列；所有行第二列
All_score = np.repeat(list(score), list(num))


def count(score):
    # 集中趋势度量
    print('均值:', np.mean(score))
    print('中位数:', np.median(score))
    print('众数:', stats.mode(score))
    # 离散趋势度量
    print('极差:', np.ptp(score))
    print('方差:', np.var(score))
    print('标准差:', np.std(score))
    print('变异系数:', np.mean(score) / np.std(score))
    # 偏度与峰度的度量
    print('偏度:', stats.skewtest(score))
    print('峰度:', stats.kurtosis(score))


count(All_score)
```

输出

均值: 57.65014855687606

中位数: 62.5

众数: ModeResult(mode=70.0, count=511)

极差: 90.0

方差: 215.64357383192066

标准差: 14.684807585798346

变异系数: 3.9258361554991996

偏度: SkewtestResult(statistic=-23.201191283130118, pvalue=4.428865440236225e-119)

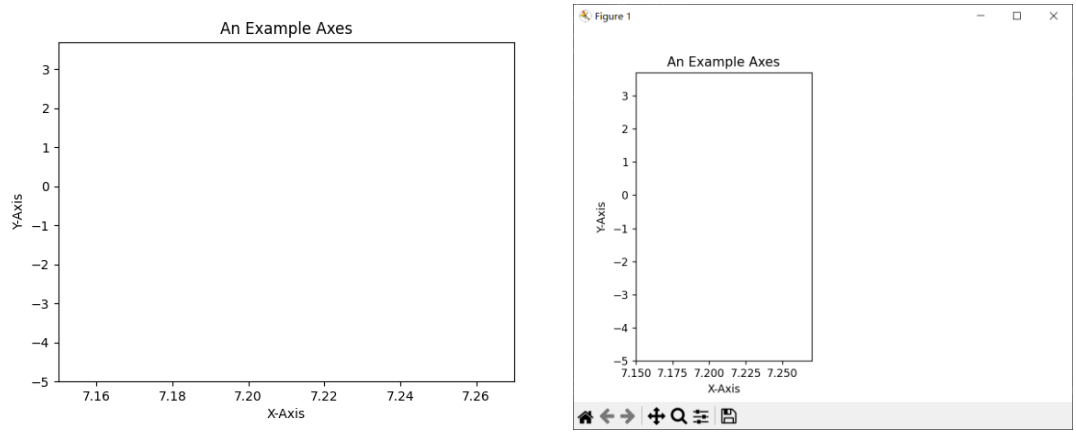
峰度: 0.7136422362619905

4.Matplotlib

绘制坐标轴

```
import matplotlib.pyplot as plt
```

```
fig = plt.figure()
ax = fig.add_subplot(111)  # 在 fig 图形对象中添加一个子图，编号为 1 行 1 列的第 1 个子图。
ax.set(xlim=[7.15, 7.27], ylim=[-5, 3.7], title='An Example Axes',
       ylabel='Y-Axis', xlabel='X-Axis')
plt.show()
```



add_subplot 用于创建和管理子图。参数改为 121 会输出右上图。

使用双语双字体

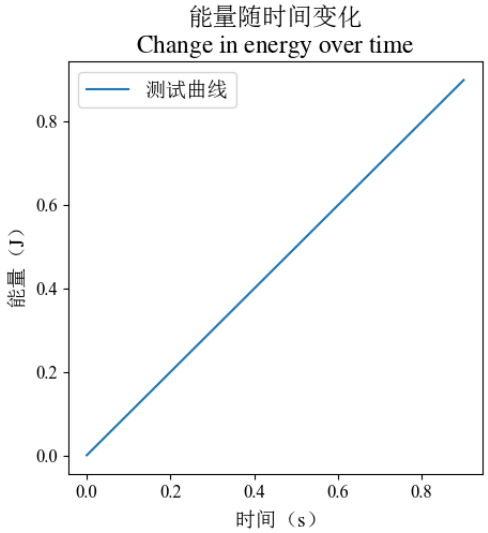
```
import matplotlib.pyplot as plt
from matplotlib import rcParams

config = {
    "font.family": 'serif',
    "font.size": 20,
    "mathtext.fontset": 'stix',
    "font.serif": ['SimSun'],
}
rcParams.update(config)

plt.title(r'宋体 $\mathrm{Times \; New \; Roman}\/\/ \alpha_i > \beta_i$')
plt.axis('off')
plt.show()
```

宋体 Times New Roman $\alpha_i > \beta_i$

如果不使用第二句 import 就要把 rcParams.update(config)改为 plt.rcParams.update(config)

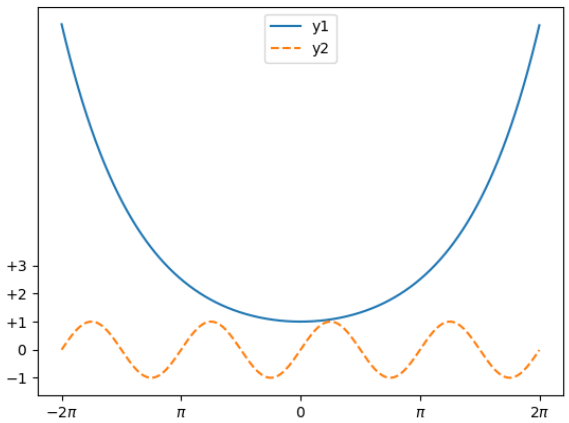
```
import matplotlib.pyplot as plt

config = {
    "font.family": "serif",  # 使用衬线体
    "font.serif": ["SimSun"],  # 全局默认使用衬线宋体
    "font.size": 14,  # 五号，10.5 磅
    "axes.unicode_minus": False,
    "mathtext.fontset": "stix",  # 设置 LaTeX 字体，stix 近似于 Times 字体
}
plt.rcParams.update(config)


fig, ax = plt.subplots(figsize=(5, 5))
ax.plot([i / 10.0 for i in range(10)], [i / 10.0 for i in range(10)])
# 中西混排，西文使用 LaTeX 罗马体
ax.set_title("能量随时间变化\n$\mathrm{Change\ in\ energy\ over\ time}$")
ax.set_xlabel("时间（$\mathrm{s}$）")
ax.set_ylabel("能量（$\mathrm{J}$）")

# 坐标系标签使用西文字体
ticklabels_style = {
```

```
    "fontname": "Times New Roman",
    "fontsize": 12,  # 小五号，9 磅
}
plt.xticks(**ticklabels_style)
plt.yticks(**ticklabels_style)
plt.legend(["测试曲线"])
plt.show()
```



绘制曲线

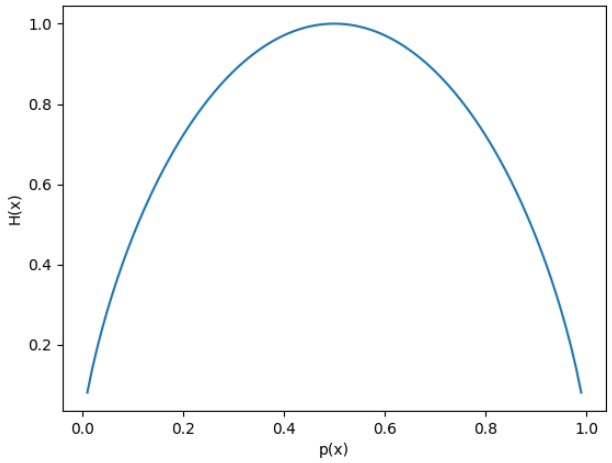
```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(-2 * np.pi, 2 * np.pi, 0.01)
y1 = np.cosh(0.5 * x)
y2 = np.sin(2 * x)
plt.plot(x, y1)
plt.plot(x, y2, '--')
plt.xticks([-2 * np.pi, -np.pi, 0, np.pi, 2 * np.pi], [r'$-2\pi$', r'$\pi$', '$0$',
'$\pi$', '$2\pi$'])
plt.yticks([-1, 0, 1, 2, 3], [r'$-1$', '$0$', '$+1$', '$+2$', '$+3$'])
plt.legend(['y1', 'y2'])
plt.show()
```



计算二元信源的熵

$$H(U) = -P\log P - (1-P)\log(1-P)$$

```python
import matplotlib.pyplot as plt
from math import log
import numpy as np


# 计算二元信息熵
def entropy(props, base=2):
    sum = 0
    for prop in props:
        sum += prop * log(prop, base)
    return sum * -1


# 构造数据
x = np.arange(0.01, 1, 0.01)
props = []
for i in x:
    props.append([i, 1 - i])

y = [entropy(i) for i in props]

plt.plot(x, y)
plt.xlabel("p(x)")
plt.ylabel("H(x)")
plt.show()
```



https://segmentfault.com/a/1190000039676856 两组离散数据求交点

https://www.cnblogs.com/vamei/archive/2012/09/17/2689798.html 饼状图

https://matplotlib.org/stable/api/index.html 库的 api 参考

https://zhuanlan.zhihu.com/p/93423829 axes 的讲解

二、应用部分

1.数据可视化

载入数据

```
import pandas as pd
import os

file_path = os.path.join('input', 'Ch3-HousePrice-train.csv')
df_train = pd.read_csv(file_path)

print("\n载入数据集：")
print(df_train)
print("\n查看数据列名：")
print(df_train.columns)
```

输出

```
载入数据集：
        Id  MSSubClass MSZoning  ...  SaleType  SaleCondition SalePrice
0        1          60       RL  ...        WD         Normal    208500
1        2          20       RL  ...        WD         Normal    181500
2        3          60       RL  ...        WD         Normal    223500
3        4          70       RL  ...        WD        Abnorml    140000
4        5          60       RL  ...        WD         Normal    250000
...    ...         ...      ...  ...       ...            ...       ...
1455  1456          60       RL  ...        WD         Normal    175000
1456  1457          20       RL  ...        WD         Normal    210000
1457  1458          70       RL  ...        WD         Normal    266500
1458  1459          20       RL  ...        WD         Normal    142125
1459  1460          20       RL  ...        WD         Normal    147500

[1460 rows x 81 columns]
```

```
查看数据列名：
Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
       'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
       'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
       'HouseStyle', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
       'RoofStyle', 'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType',
       'MasVnrArea', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual',
       'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinSF1',
       'BsmtFinType2', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', 'Heating',
       'HeatingQC', 'CentralAir', 'Electrical', '1stFlrSF', '2ndFlrSF',
       'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath',
       'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual',
       'TotRmsAbvGrd', 'Functional', 'Fireplaces', 'FireplaceQu', 'GarageType',
       'GarageYrBlt', 'GarageFinish', 'GarageCars', 'GarageArea', 'GarageQual',
       'GarageCond', 'PavedDrive', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'PoolQC',
       'Fence', 'MiscFeature', 'MiscVal', 'MoSold', 'YrSold', 'SaleType',
       'SaleCondition', 'SalePrice'],
      dtype='object')
```

为什么使用 os.path.join:

正斜杠/是在类 Unix 系统（如 Linux 和 macOS）上使用的路径分隔符。反斜杠\是在

Windows 系统上使用的路径分隔符。对于跨平台的 Python 代码，建议使用 os 模块提供的函数来处理文件路径，这样可以保证在不同的操作系统上都能正常工作。例如，使用 os.path.join()函数可以正确地创建跨平台的文件路径，它会根据当前操作系统自动选择正确的路径分隔符。

描述数据

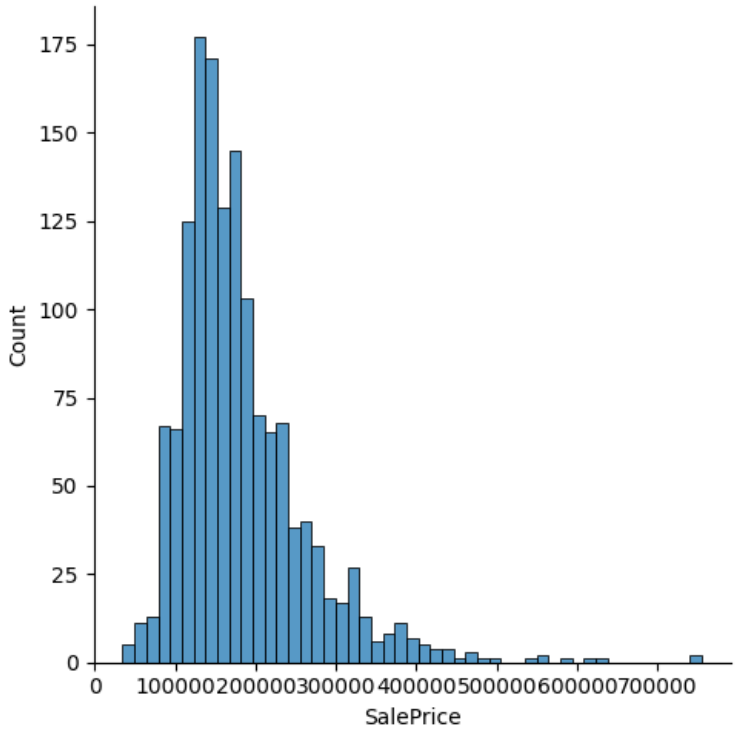
```
print(df_train['SalePrice'].describe())
```

```
count      1460.000000
mean     180921.195890
std       79442.502883
min       34900.000000
25%      129975.000000
50%      163000.000000
75%      214000.000000
max      755000.000000
Name: SalePrice, dtype: float64
```

绘制图像

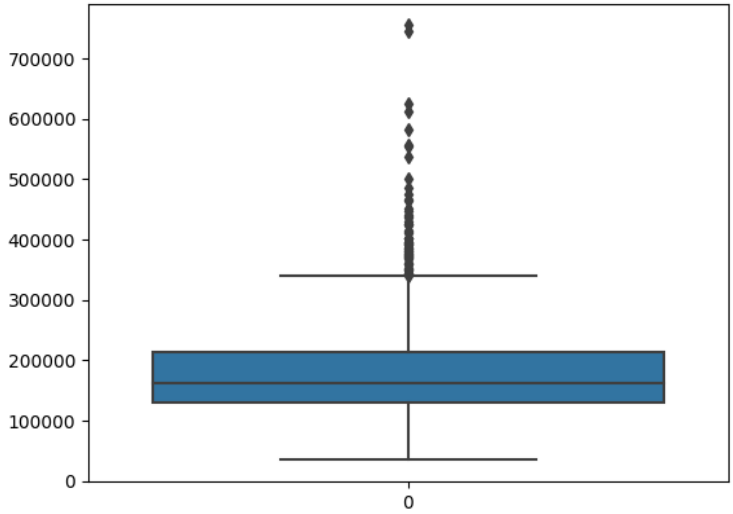
直方图

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.displot(df_train['SalePrice'])  # 直方图
plt.show()
```



箱型图

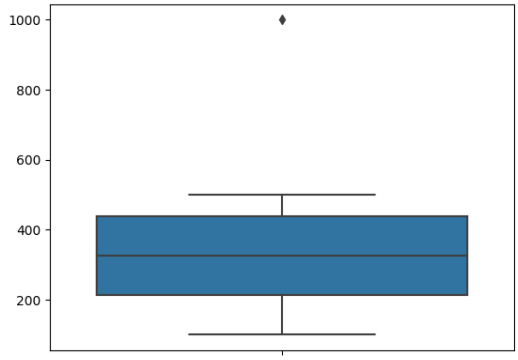
```
sns.boxplot(df_train['SalePrice'])  # 箱型图
plt.show()
```



为计算方便，这里使用一个数目较少的数据集来说明一下箱型图：

```
prices = [100, 150, 200, 250, 300, 350, 400, 450, 500, 1000]
df = pd.DataFrame(prices, columns=['Price'])
summary_stats = df.describe().transpose()
print(summary_stats)
sns.boxplot(y=prices)
plt.show()
```

```
       count   mean         std    min     25%    50%    75%     max
Price   10.0  370.0  256.255081  100.0  212.5  325.0  437.5  1000.0
```



由输出数据可知：总数=10，均值=370，标准差=256.255，Q1=200+(250-200)*25%=212.5，中位数 Q2=(300+350)/2=325.0，Q3=400+(450-400)*0.75%=437.5

计算得：

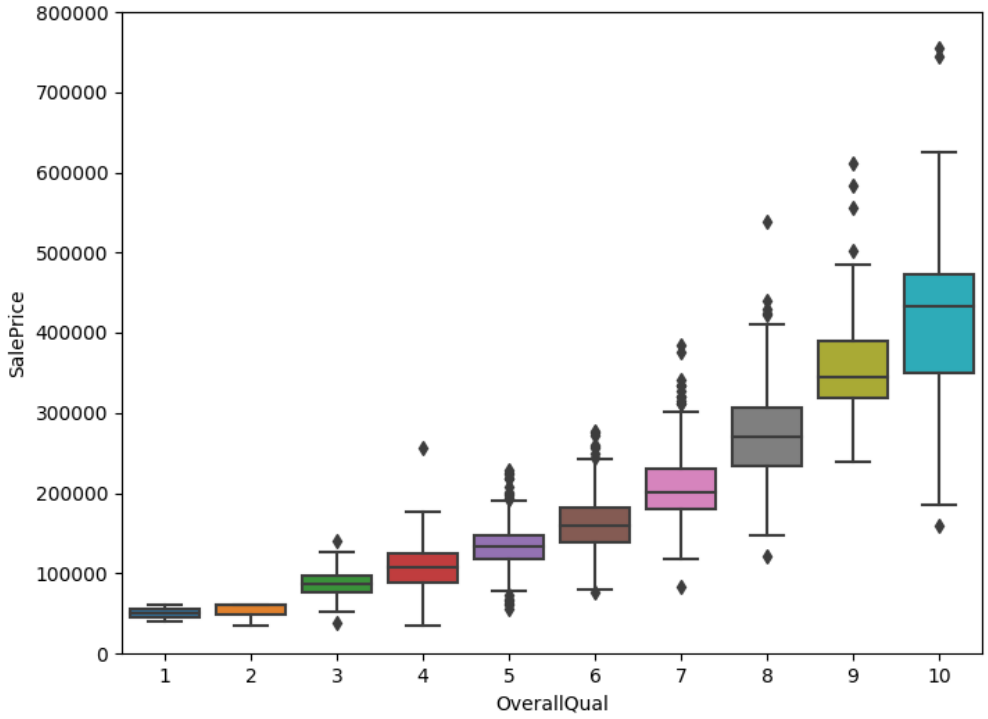
IQR=Q3-Q1=225。

箱型图中对应的五条线依次是：

下限=MIN=100，Q1=212.5，Q2=325.0，Q3=437.5，上限=去掉异常值 1000 后的最大值=500。

一组箱型图

```
data = pd.concat([df_train['SalePrice'], df_train['OverallQual']], axis=1)
f, ax = plt.subplots(figsize=(8, 6))  # 创建画像 figure 与一组子图 subplots
fig = sns.boxplot(x='OverallQual', y='SalePrice', data=data)
fig.axis(ymin=0, ymax=800000)
plt.show()
```



语法：

① **new_dataframe = pd.concat([dataframe1, dataframe2], axis=0/1)**。

dataframe1 和 dataframe2：要合并的数据集，可以是 pandas DataFrame 或 Series 的列表。

axis：指定合并的方向。axis=0 表示按行合并(将两个 DataFrame 垂直合并，行索引相同的列会对齐)，axis=1 表示按列合并(将两个 DataFrame 水平合并，列索引相同的列会对齐)。

```
import pandas as pd
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df2 = pd.DataFrame({'B': [7, 8, 9], 'C': [10, 11, 12]})
merged_rows = pd.concat([df1, df2], axis=0)
merged_columns = pd.concat([df1, df2], axis=1)
print(df1)
print(df2)
print(merged_rows)
print(merged_columns)
```

如下图，分别为 df1,df2,按行处理,按列处理的结果。注意没有对应的列的值被填充为缺失值 NaN。

	A	B
0	1	4
1	2	5
2	3	6

	B	C
0	7	10
1	8	11
2	9	12

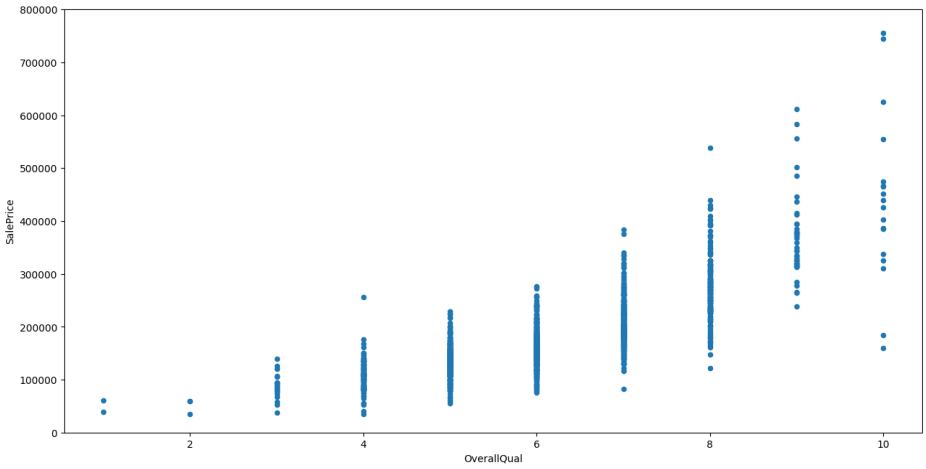
	A	B	C
0	1.0	4	NaN
1	2.0	5	NaN
2	3.0	6	NaN
0	NaN	7	10.0
1	NaN	8	11.0
2	NaN	9	12.0

	A	B	B	C
0	1	4	7	10
1	2	5	8	11
2	3	6	9	12

② (8, 6)用于设置画布（figure）的大小，宽度为 8 英寸，高度为 6 英寸。

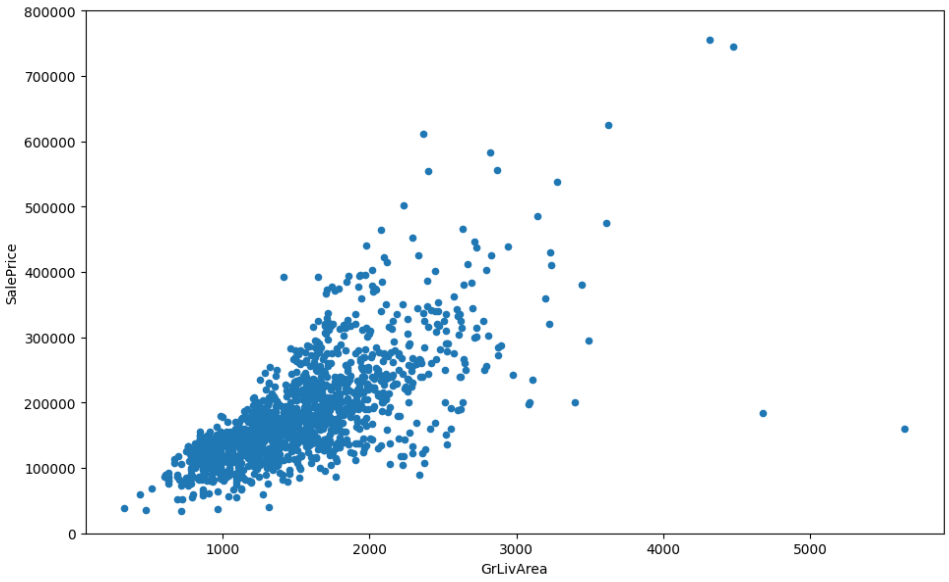
散点图

```
data = pd.concat([df_train['SalePrice'], df_train['OverallQual']], axis=1)
data.plot.scatter(x='OverallQual', y='SalePrice', ylim=(0, 800000))  # 散点图
plt.show()
```



该散点图显示的不够好看，因为参数 x 的值过少。我们可以把参数 OverallQual(总体质量)换成 GrLivArea(平方英尺)，此时生成的散点图就与平时常见的类似了。

```
data = pd.concat([df_train['SalePrice'], df_train['GrLivArea']], axis=1)
data.plot.scatter(x='GrLivArea', y='SalePrice', ylim=(0, 800000))
plt.show()
```



2.数据清洗

缺失值处理

```
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer

df = pd.DataFrame(np.random.randn(6, 4), columns=list('abcd'))  # 随机生成数据
df.iloc[4, 3] = np.nan  # 选择第 5 行、第 4 列
df.loc[3] = np.nan  # 选择第 4 行
print(df)

total = df.isnull().sum().sort_values(ascending=False)
percent = (df.isnull().sum() / df.isnull().count().sort_values(ascending=False))
missing_data = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])
print(missing_data)  # 查看缺失值的数量

df_cleaned = df.dropna(how='all')  # 删除全为缺失值的行
print(df_cleaned)

imp = SimpleImputer(missing_values=np.nan, strategy='mean')
imp.fit(df)
df_cleaned_1 = imp.transform(df_cleaned)
print(df_cleaned_1)
```

输出依次为 df,missing_data,df_cleaned,df_cleaned_1

```
          a         b         c         d
0  1.254260  0.232693  1.631425 -0.053841
1 -0.462248  0.360201  0.386029  0.312850
2  1.500807  0.796929  1.392375 -1.914533
3       NaN       NaN       NaN       NaN
4  0.642735 -1.030269 -0.397600       NaN
5  0.962220  0.147848 -0.625509  0.710948
```

```
   Total   Percent
d      2  0.333333
a      1  0.166667
b      1  0.166667
c      1  0.166667
```

```
          a         b         c         d
0  1.254260  0.232693  1.631425 -0.053841
1 -0.462248  0.360201  0.386029  0.312850
2  1.500807  0.796929  1.392375 -1.914533
4  0.642735 -1.030269 -0.397600       NaN
5  0.962220  0.147848 -0.625509  0.710948
```

```
[[ 1.25425977  0.23269252  1.63142545 -0.05384106]
 [-0.46224823  0.36020057  0.38602899  0.31284953]
 [ 1.50080682  0.79692885  1.39237459 -1.91453265]
 [ 0.64273546 -1.03026891 -0.39760032 -0.23614407]
 [ 0.96221962  0.14784837 -0.62550899  0.71094789]]
```

注：

(1)df.iloc 和 df.loc 是 Pandas 中用于访问和选择 DataFrame 数据的两种方法，区别在于索引方式不同：①df.iloc 使用**整数**位置索引，通过指定行号和列号来选择数据。例如，df.iloc[0, 0]代表选择第一行第一列的元素。②df.loc 使用**标签**索引，通过指定行标签和列标签来选择数据。比如：df.loc[0,'column_name']代表选择行标签为 0、列标签为'column_name'的元素。

(2)df.isnull().sum()返回一个 Series 对象，其中每列的标签作为索引，对应的值是该列中缺失值的数量。sort_values(ascending=False)用于对 Series 进行降序排列。参数 ascending=False 表示按递减顺序(即：从大到小)排列。

(3)df.dropna(how='all')中的 how 参数表示删除的条件。how='all'表示只删除全为缺失值的行，如果一行中所有元素都是缺失值，那么该行才将被删除。默认值 how='any'表示删除含有任意缺失值的行，只要有一个元素是缺失值，该行就会被删除。

(4)imp = SimpleImputer(missing_values=np.nan, strategy='mean')用于处理缺失值。SimpleImputer 是 scikit-learn 库中的一个类，用于使用指定的策略填充缺失值。missing_values=np.nan 表示缺失值的标识，默认为 np.nan。strategy='mean'表示使用均值作为策略来填充缺失值。其他可选的策略包括 'median'（中位数）和 'most_frequent'（众数）。

(5)imp.fit(df)是用于拟合填充器对象 imp 的方法。通过调用 fit 方法，填充器会根据指定的策略和数据集 df 来学习缺失值的填充规则。在这个过程中，df 并没有被修改，而是用于计算填充所需的统计信息。fit 方法没有返回值，但是填充器对象 imp 本身被修改以

存储学习到的规则。

(6)df_cleaned_1 = imp.transform(df_cleaned)是用于将填充器应用于数据集的语法，transform 方法会根据学习到的规则填充数据集中的缺失值。在这里，参数 df_cleaned 是传递给 transform 方法的数据集，而不是原始的 df，因为 df_cleaned 中不包含全为缺失值的行。transform 方法返回填充后的数据集，并存储在 df_cleaned_1 中。

举一个简单的例子来说明 SimpleImputer:

```
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer

data = {'A': [7, 1, 5],
        'B': [7, 2, 7],
        'C': [1, 2, 3],
        'D': [6, 4, 8]}
df = pd.DataFrame(data)  # 使用字典创建数据表
df.iloc[1, 2] = np.nan  # 选择行、列
print("df(with NaN):")
print(df)

total = df.isnull().sum().sort_values(ascending=False)
percent = (df.isnull().sum() / df.isnull().count().sort_values(ascending=False))
missing_data = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])
print("missing_data:")
print(missing_data)  # 查看缺失值的数量

imp = SimpleImputer(missing_values=np.nan, strategy='mean')
imp.fit(df)
df_fill = imp.transform(df)
print("df_fill:")
print(df_fill)
```

输出

```
df(with NaN):
   A  B    C  D
0  7  7  1.0  6
1  1  2  NaN  4
2  5  7  3.0  8
```

```
missing_data:
   Total   Percent
C      1  0.333333
A      0  0.000000
B      0  0.000000
D      0  0.000000
```

```
df_fill:
[[7. 7. 1. 6.]
 [1. 2. 2. 4.]
 [5. 7. 3. 8.]]
```

这是使用的是均值填补，也就是取该列其他值的平均数：(1+3)/2=2。

使用中位数，众数，常量(不指定，默认为 0)，常量(指定为 1)

```
imp = SimpleImputer(missing_values=np.nan, strategy="median")
imp = SimpleImputer(missing_values=np.nan, strategy="most_frequent")
imp = SimpleImputer(missing_values=np.nan, strategy="constant")
imp = SimpleImputer(missing_values=np.nan, strategy="constant", fill_value=1)
```

得到 2 1 0 1

噪声平滑

首先创建带有噪声的函数：

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

t = np.linspace(0, 2 * np.pi, 500)  # 创建一个包含 500 个点的时间数组 t，范围是从 0 到 2π
y = 100 * np.sin(t)  # y = 100sint
```

```
noise = np.random.normal(0, 15, 500)  # 创建一个服从正态分布的，共 500 个值的噪声数组，均值为 0，标准差为 15
y = y + noise  # 添加噪声
plt.figure()  # 创建一个新的图形窗口
plt.plot(t, y)  # 绘制时间 t 与带有噪声的信号 y 之间的关系曲线
plt.xlabel('t')
plt.ylabel('y = sin(t) + noise')
plt.show()  # 显示图形
```

移动平均 Moving Average:

```
window_size = 10  # 窗口大小
y_smoothed = np.convolve(y, np.ones(window_size) / window_size, mode='same')  # mode='same'表示保持：与原始信号相同的长度
plt.figure()
plt.plot(t, y_smoothed)
plt.xlabel('t')
plt.ylabel('Smoothed y (Moving Average)')
plt.show()
```

中值滤波 Median Filtering:

```
from scipy.signal import medfilt
window_size = 11  # 窗口大小
y_smoothed = medfilt(y, kernel_size=window_size)  # 中值滤波器的窗口大小 kernel_size 要求是奇数
plt.figure()
plt.plot(t, y_smoothed)
plt.xlabel('t')
plt.ylabel('Smoothed y (Median Filtering)')
plt.show()
```

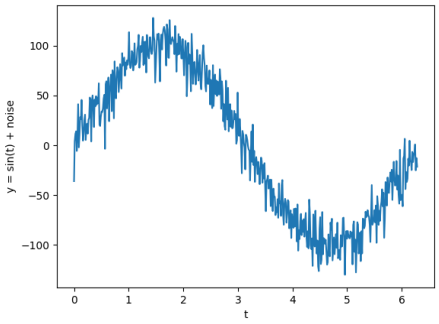
加权平均 Weighted Averaging:

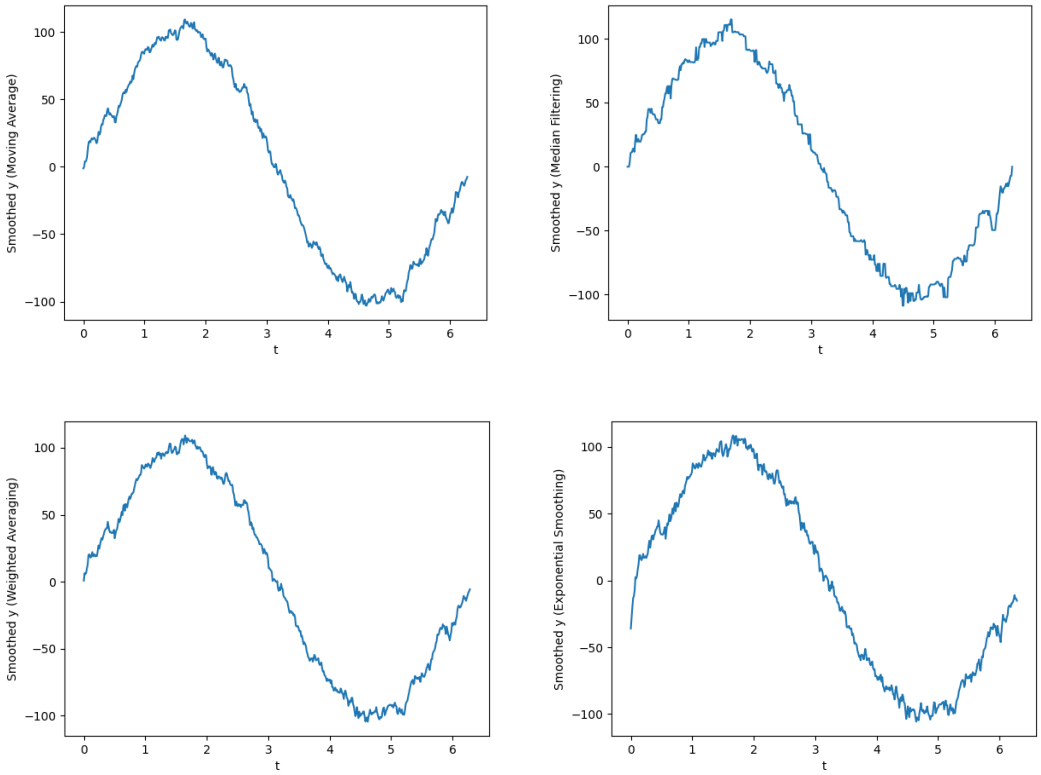
```
weights = np.exp(-np.arange(window_size) / window_size)  # 离中心位置越远的权重越小
weights /= np.sum(weights)  # 进行除法运算后的 weight 总和为 1
y_smoothed = np.convolve(y, weights, mode='same')
plt.figure()
plt.plot(t, y_smoothed)
plt.xlabel('t')
plt.ylabel('Smoothed y (Weighted Averaging)')
plt.show()
```

指数平滑 Exponential Smoothing:

```
alpha = 0.2  # 数据波动大，平滑系数 α 取大一些。数据波动小就取小一些
y_smoothed = [y[0]]
for i in range(1, len(y)):
    y_smoothed.append(alpha * y[i] + (1 - alpha) * y_smoothed[i - 1])
plt.figure()
plt.plot(t, y_smoothed)
plt.xlabel('t')
plt.ylabel('Smoothed y (Exponential Smoothing)')
plt.show()
```

输出如下图:





然后来详细说说移动平均中的卷积（因为其他三个理解起来比较简单）：

移动平均：

convolve 计算的是 y 与[np.ones(window_size) / window_size]的卷积。

你应该知道 y 是 500 个数的数组，而 np.ones(window_size) / window_size =[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]是 10 个数的数组。

那**什么是卷积**？

先说一个奇妙的比喻：如果你收到一个原味鸡肉卷，但又想吃酸甜口的鸡肉卷，那就打开鸡肉卷，把色拉酱均匀的淋下去，重新卷回来，你就可以美味享用了。

你可以把卷积想象成：把毛巾沿着某条线卷起来。卷起来后某点 A 处的函数值正好为这条直线上函数值的积分（这条直线即为 A 点在卷的时候沿卷的方向形成的线）。

而所谓两个函数的卷积，本质上就是先将一个函数翻转，然后进行滑动叠加。

因为卷积是为了表示一个函数对另一个函数所有微量的响应的总叠加效果。如果不翻转过来，叠加出来结果的**时序**就是反的，所以就人为规定把其中一个信号翻转过来。

Therefore, the operation is in some sense the “revolving” of one of the input functions with the other one(从某种意义上说，操作是将一个输入函数与另一个输入函数进行某种形式的"旋转"). Particularly in mathematics, physics, and related areas, the verb commonly used to designate such a revolving is “to convolve.” This verb comes from the Latin words con and volve˘re, which mean “**together**” and “**roll up**,” respectively; thus, convolve means “**roll up together**.” Accordingly, the action of convolving is called convolution.

卷积的数学定义：

称 $(f*g)(n)$ 为 f,g 的卷积（翻转积分/褶积）。

其连续的定义为：$(f*g)(n)=\int_{-\infty}^{\infty} f(\tau)g(n-\tau)\mathrm{d}\tau$

其离散的定义为：$(f*g)(n)=\sum_{\tau=-\infty}^{\infty} f(\tau)g(n-\tau)$

一些例子：

离散型的例子：骰子之和为 4 的概率

记 $f(n), g(n)$ 为投到 n 的概率， $P = f(1)g(3)+f(2)g(2)+f(3)g(1) = \sum_{m=1}^{3} f(4-m)g(m) = (f*g)(4)$

连续型的例子：馒头的腐败

记馒头生产速度为 $f(n)$，腐败函数为 $g(n)$，则 24 小时后馒头总共腐败了 $\int_0^{24} f(t)g(24-t)\mathrm{d}t$，因为在 t 时刻产出的馒头只回腐败 $24-t$ 小时。（这就是为什么卷积要翻转一个函数，因为是物理意义给出的）

图像处理：

记 $g = \begin{bmatrix} b_{-1,-1} & b_{-1,0} & b_{-1,1} \\ b_{0,-1} & b_{0,0} & b_{0,1} \\ b_{1,-1} & b_{1,0} & b_{1,1} \end{bmatrix}$，新矩阵为 $c = \begin{bmatrix} c_{0,0} & \cdots & c_{0,n} \\ \vdots & \ddots & \vdots \\ c_{m,0} & \cdots & c_{m,n} \end{bmatrix}$，取 $f = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix}$，则有：

$$c_{1,1} = f*g = a_{0,0}b_{1,1}+a_{0,1}b_{1,0}+a_{0,2}b_{1,-1} + a_{1,0}b_{0,1}+a_{1,1}b_{0,0}+a_{1,2}b_{0,-1} + a_{2,0}b_{-1,1}+a_{2,1}b_{-1,0}+a_{2,2}b_{-1,1},$$

即：$(f*g)(1,1) = \sum_{k=0}^{2}\sum_{h=0}^{2} f(h,k)g(1-h,1-k)$

二维离散形式的卷积公式：$(f*g)(u,v) = \sum_i \sum_j f(i,j)g(u-i,v-j) = \sum_i \sum_j a_{i,j}b_{u-i,v-j}$

对于一般的计算方法：

①在原始图像上取出点 (u,v) 处的矩阵（大小与 g 同型）：$f = \begin{bmatrix} a_{u-1,v-1} & a_{u-1,v} & a_{u-1,v+1} \\ a_{u,v-1} & a_{u,v} & a_{u,v+1} \\ a_{u+1,v-1} & a_{u+1,v} & a_{u+1,v+1} \end{bmatrix}$

②将图像处理矩阵 $g = \begin{bmatrix} b_{-1,-1} & b_{-1,0} & b_{-1,1} \\ b_{0,-1} & b_{0,0} & b_{0,1} \\ b_{1,-1} & b_{1,0} & b_{1,1} \end{bmatrix}$ 沿 x 轴翻转后沿 y 轴翻转得 $g' = \begin{bmatrix} b_{1,1} & b_{1,0} & b_{1,-1} \\ b_{0,1} & b_{0,0} & b_{0,-1} \\ b_{-1,1} & b_{-1,0} & b_{-1,-1} \end{bmatrix}$

③卷积即为 f, g' 的内积：$f*g(u,v) = a_{u-1,v-1}b_{1,1}+a_{u-1,v}b_{1,0}+a_{u-1,v+1}b_{1,-1} + a_{u,v-1}b_{0,1}+a_{u,v}b_{0,0}+a_{u,v+1}b_{0,-1} + a_{u+1,v-1}b_{-1,1}+a_{u+1,v}b_{-1,0}+a_{u+1,v+1}b_{-1,-1}$

事实上，数学中的卷积和卷积神经网络 CNN 中的卷积严格意义上是两种不同的运算。

数学中卷积，主要是为了诸如信号处理、求两个随机变量和的分布等而定义的运算，所以需要“翻转”是根据问题的需要而确定的。

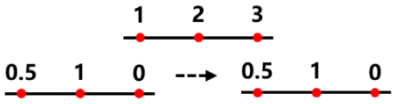
卷积神经网络中“卷积”，是为了提取图像的特征，其实只借鉴了“**加权求和**”的特点。

数学中的“卷积核”都是已知的或者给定的，卷积神经网络中“卷积核”本来就是 trainable 的参数，不是给定的，根据数据训练学习的，那么翻不翻转还有什么关系呢？因为无论翻转与否对应的都是未知参数！

具体图案可以参考：https://mlnotebook.github.io/post/CNN1/

计算举例：np.convolve([1, 2, 3], [0, 1, 0.5])

计算方法 1：将其中一个数组反向，将两个数组上下排放，然后上下对应的项相乘相加，如右图。



计算过程：

①0.5,1,0 在最左边，只有 0 与 1 对齐了，1*0=0　②0.5,1,0 往右平移一格，1*1+2*0=1

③右平移，1*0.5+2*1+3*0=2.5　④右平移，2*0.5+3*1=4　⑤右平移，3*0.5=1.5

答案：[0.　1.　2.5 4.　1.5]

计算方法 2：视作 x^2+2x+3 与 $0 \cdot x^2+x+0.5$ 相乘，得到 $0 \cdot x^4+x^3+2.5x^2+4x+1.5$

比如：

```
y = [0, 1, 3, 2, 3, 2, 3, 1, 4, 3, 4, 3]
print(np.convolve(y, [0.2, 0.2, 0.2, 0.2, 0.2]))
print(np.convolve(y, [0.2, 0.2, 0.2, 0.2, 0.2], mode='same'))
```

输出

```
[0.  0.2 0.8 1.2 1.8 2.2 2.6 2.2 2.6 2.6 3.  3.  2.8 2.  1.4 0.6]
[0.8 1.2 1.8 2.2 2.6 2.2 2.6 2.6 3.  3.  2.8 2. ]
```

将第二行与 y 相比

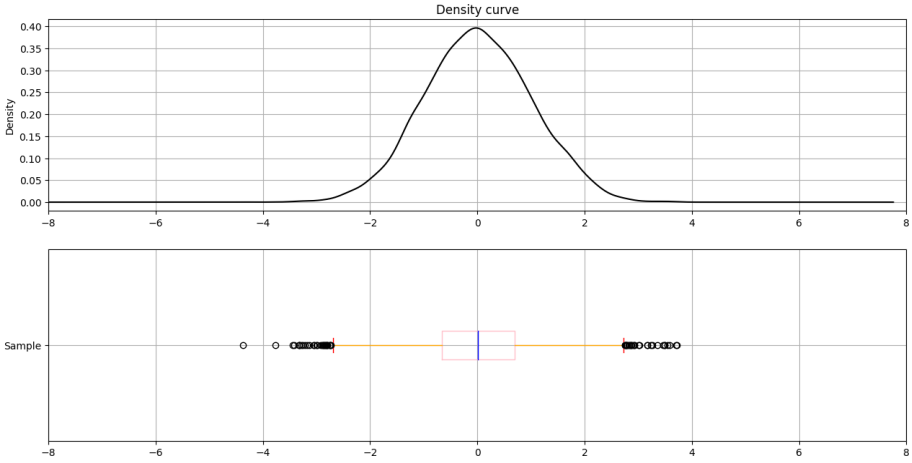
```
[0   1   3   2   3   2   3   1   4   3   4   3]
```

明显缓和了很多。

异常值的识别

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats

data = pd.Series(np.random.randn(10000))
u = data.mean()
std = data.std()
stats.kstest(data, 'norm', (u, std))
print("均值:%.3f,标准差:%.3f" % (u, std))

fig = plt.figure(figsize=(10, 6))
ax1 = fig.add_subplot(2, 1, 1)
data.plot(kind='kde', grid=True, style='-k', title='Density curve')
ax1.set_xlim(-8, 8)

error_std = data[np.abs(data - u) > 3 * std]
data_c_std = data[np.abs(data - u) <= 3 * std]
print("使用 3std 原则得到的异常值个数:%d" % (len(error_std)))

ax2 = fig.add_subplot(2, 1, 2)
color = dict(boxes='Pink', whiskers='Orange', medians='Blue', caps='Red')
data.plot.box(vert=False, grid=True, color=color, ax=ax2, label='Sample')
ax2.set_xlim(-8, 8)
plt.show()

q1 = data.quantile(0.25)
q3 = data.quantile(0.75)
IQR = q3 - q1
MIN = q1 - 1.5 * IQR
MAX = q3 + 1.5 * IQR
print("分位差:%.3f,下限:%.3f,上限:%.3f" % (IQR, MIN, MAX))

error_iqr = data[(data < MIN) | (data > MAX)]
data_c = data[(data <= MIN) & (data <= MAX)]
print("使用 IQR 原则得到的异常值个数:%d" % (len(error_iqr)))
```

图片输出



文字输出

```
均值:0.017,标准差:1.004
使用 3std 原则得到的异常值个数:27
分位差:1.360,下限:-2.703,上限:2.736
使用 IQR 原则得到的异常值个数:58
```

语法：

①stats.kstest(data, 'norm', (u, std))

data：输入的数据数组或序列，用于进行假设检验。

'norm'： 进行假设检验的分布函数，这里是正态分布。'uniform' 表示均匀分布、'expon' 表示指数分布等。

(u, std)：指定分布函数(这里是正态分布)的参数值，其中 u 是均值，std 是标准差。

该函数使用 Kolmogorov-Smirnov 检验来比较输入数据与指定的概率分布函数之间的拟合程度。输出如 KstestResult(statistic=0.006038808990869726, pvalue=0.8568670428232958, statistic_location=0.9815395126873121, statistic_sign=-1)所示。

②data.plot(kind='kde', grid=True, style='-k', title='Density curve')

kind='kde'：指定绘制的图形类型为核密度估计图（Kernel Density Estimate），该图形可以用来估计数据的概率密度函数。

grid=True：在图形中显示网格线。

style='-k'：设置曲线的样式为实线'-'且颜色为黑色'k'。

title='Density curve'：设置图形的标题。

图形绘制时，使用核密度估计方法对数据进行平滑处理，从而得到数据的概率密度曲线图 Density curve。

③data[np.abs(data - u) > 3 * std]是根据条件索引，选取满足条件的数据值构成一个新的数组 error_std。

3.数据集成

元组重复

```
import numpy as np
import pandas as pd
from scipy import stats

df = pd.read_csv('input/Ch3-HousePrice-train.csv')
YearBuilt = df.YearBuilt
YearRemodAdd = df.YearRemodAdd
print(stats.chisquare(YearBuilt, f_exp=YearBuilt))  # just 为了输出

# 将房屋面积和售价连接为 df_Area_Price
GrLivArea = df.GrLivArea
SalePrice = df.SalePrice
Area_Price = np.array([GrLivArea, SalePrice])
df_Area_Price = pd.DataFrame(Area_Price.T, columns=['GrLivArea', 'SalePrice'])

covariance = df_Area_Price.GrLivArea.cov(df_Area_Price.SalePrice)
print(covariance)
correlation = df_Area_Price.GrLivArea.corr(df_Area_Price.SalePrice)
print(correlation)
```

输出

```
Power_divergenceResult(statistic=0.0, pvalue=1.0)
29581866.743236598
0.7086244776126521
```

语法：scipy.stats.chisquare(f_obs, f_exp=None, ddof=0, axis=0)

参数：f_obs 观测频率，f_exp 预期频率(默认值 the categories are equally likely)

返回值：statistic：卡方统计量，度量观测值与期望值之间的**偏离程度**。较大的统计量说明存在显著的差异，反之则意味着观测值与期望值非常接近，没有显著的差异。

pvalue：卡方检验的 p 值，用于确定观测值与期望值之间的**差异是否显著**。如果 pvalue 非常小（通常小于显著性水平，例如 0.05）表示差异是显著的，我们可以拒绝零假设。反之则不能拒绝零假设，表示没有足够的证据表明观测值与期望值之间存在显著差异。

书中使用 print(stats.chisquare(YearBuilt, f_exp=YearRemodAdd))会报错 ValueError: For each axis slice, the sum of the observed frequencies must agree with the sum of the expected frequencies to a relative tolerance of 1e-08, but the percent differences are:0.006898070951487656(对于每个 axis ，f_obs 的总和与 f_exp 的总和必须一致(容忍量为 1e-08))，因为可以通过检验协方差和相关系数=369.67545607330976，0.5928549763436504，由此说明这两个其实偏离程度很大。

协方差 cov 测量两个变量之间的总体线性关系。正协方差表示正相关，负协方差表示负相关，协方差接近零表示变量之间没有明显的线性关系。协方差的值的大小并不容易解释，因为它受到变量单位的影响。

相关系数 corr 是协方差的标准化版本，衡量两个变量之间的线性关系的**强度和方向**。接近 1 或-1，表示存在强烈的线性关系，接近 0 表示两个变量之间几乎没有线性关系。

4.数据归约

PCA 与属性子集选择

```
import numpy as np
import pandas as pd
from sklearn.feature_selection import RFE
from sklearn.linear_model import LinearRegression
from sklearn.decomposition import PCA

data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep=r"\s+", skiprows=22, header=None)  # 跳过了文件的前 22 行，没有使用文件的第一行作为列名

# data 包含了特征数据，而 target 包含了目标数据。
# 因为原数据集是从 23 行开始，每两行对应一行的数据。
# 所以 data 选择 raw_df 中的奇数行（raw_df.values[::2, :]）和偶数行的前两列（raw_df.values[1::2, :2]）
data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
# target 由 raw_df 中的偶数行的第三列（raw_df.values[1::2, 2]）组成
target = raw_df.values[1::2, 2]
print(data.shape)

pca = PCA()
pca.fit(data)
print(pca.explained_variance_ratio_)  # 输出方差百分比

pca = PCA(3)
pca.fit(data)
low_d = pca.transform(data)  # 降维
print(low_d.shape)

names = ["CRIM", "ZN", "INDUS", "CHAS", "NOX", "RM", "AGE", "DIS", "RAD", "TAX", "PTRATIO", "B", "LSTAT", "MEDV"]
IR = LinearRegression()  # 创建了一个线性回归模型的实例
# 创建了一个递归特征消除（RFE）对象
rfe = RFE(IR, n_features_to_select=1)  # 使用指定的模型(线性回归模型 IR)来选择 n_features_to_select 个最重要的特征。
rfe.fit(data, target)  # 使用 RFE 对象拟合数据集和目标变量。RFE 将使用线性回归模型来评估每个特征的重要性，并选择最重要的特征。
# 将 RFE 对象的特征排名(ranking)与特征名称进行组合，然后对其进行排序。
# map(lambda x: round(x, 4), rfe.ranking_) 用于将排名四舍五入到小数点后四位。最后，将排名和特征名称组合在一起并打印出来。
print(sorted(zip(map(lambda x: round(x, 4), rfe.ranking_), names)))
```

输出

```
(506, 13)
[8.05823175e-01 1.63051968e-01 2.13486092e-02 6.95699061e-03
 1.29995193e-03 7.27220158e-04 4.19044539e-04 2.48538539e-04
 8.53912023e-05 3.08071548e-05 6.65623182e-06 1.56778461e-06
 7.96814208e-08]
(506, 3)
[(1, 'NOX'), (2, 'RM'), (3, 'CHAS'), (4, 'PTRATIO'), (5, 'DIS'), (6, 'LSTAT'), (7, 'RAD'), (8, 'CRIM'), (9, 'INDUS'), (10, 'ZN'), (11, 'TAX'), (12, 'B'), (13, 'AGE')]
```

从方差百分比可以看出前三维的贡献率达到 99%以上(8.05823175e-01+1.63051968e-01+2.13486092e-02)。可以用来判断应该选取几个主成分。

在递归特征消除（RFE）中，目标是识别哪些特征对模型预测的贡献最大。从全集开始，每次删除当前最坏的属性，剩下的特征将被用于重新训练模型。以此类推，每次迭代都会删

除一个最不重要的特征，直到所有特征都被考虑。因此，RFE 的排名指的是特征被删除的顺序。排名 1 的特征是最后一个被考虑删除的，也就是说它是最重要的。

数据集描述：

1- CRIM	犯罪率	2- ZN	超过 25,000 sq.ft.的住宅用地比例
3- INDUS	非零售商业用地占比	4- CHAS	是否临 Charles 河
5- NOX	氮氧化物浓度	6- RM	房屋房间数
7- AGE	房屋年龄	8- DIS	和就业中心的距离
9- RAD	是否容易上高速路	10- TAX	税率
11- PTRATIO	学生人数比老师人数	12- B	城镇黑人比例计算的统计值
13- LSTAT	低收入人群比例	14- MEDV	房价中位数

5.数据变换

数据规范化

先说一个 bug 的解决方法：

```
original_data1 = [-2, -1, 0, 1, 2, 3, 4]
original_data1 = np.array(original_data1)
min_val = original_data1.min
max_val = original_data1.max
min_max_normalized = (original_data1 - min_val) / (max_val - min_val)
```

报错 TypeError: unsupported operand type(s) for -: 'int' and 'builtin_function_or_method'，并且输出 min_val 是<built-in method min of numpy.ndarray object at 0x000001F2666F51D0>

这是因为省略了 min 的()会导致 Python 将其**解释为属性而不是函数调用**。

前置知识：函数加括号是指对此**函数**的调用，函数不加括号是指调用函数本身（的**内存地址**），也可以理解成对函数重命名。

通过以下代码很容易知道区别：

```
def greet(name):
    return f"Hello, {name}!"

# 函数加括号是函数调用，返回函数执行结果
message = greet("A")  # 调用函数 greet，将返回值赋给变量 message
print(message)  # 输出 "Hello, A!"

# 函数不加括号，将函数本身赋值给一个变量
greeting_function = greet
# 此时 greeting_function 包含了函数 greet 的引用

# 使用函数引用来调用函数
message = greeting_function("B")  # 调用 greeting_function，其实是调用 greet
print(message)  # 输出 "Hello, B!"
```

同理：

```
a = min_val()
print(a)  # 输出 -2
```

因为 min_val 其实就是 original_data1.min 了，original_data1.min()理所当然的就是-2。

各种规范化方法的实现：

```
import numpy as np


class DataNorm:
    def __init__(self, input_array):
        self.arr = np.array(input_array)
        self.x_max = self.arr.max(initial=None)  # 最大值
        self.x_min = self.arr.min(initial=None)  # 最小值
        self.x_mean = self.arr.mean()  # 平均值
        self.x_std = self.arr.std()  # 标准差
        self.decimals_setting = 3  # 小数位数

    def Min_MaxNorm(self):
        arr = np.around(((self.arr - self.x_min) / (self.x_max - self.x_min)),
decimals=self.decimals_setting)
        print("Min_Max 标准化:{}".format(arr))

    def Z_ScoreNorm(self):
        arr = np.around((self.arr - self.x_mean) / self.x_std,
decimals=self.decimals_setting)
        print("Z_Score 标准化:{}".format(arr))

    def Decimal_ScalingNorm(self):
        power = 1
```

```
        maxValue = self.x_max
        while maxValue / 10 >= 1.0:
            power += 1
            maxValue /= 10
        arr = np.around((self.arr / pow(10, power)), decimals=self.decimals_setting)
        print("小数定标标准化:{}".format(arr))

    def MeanNorm(self):
        first_arr = np.around((self.arr - self.x_mean) / (self.x_max - self.x_min),
decimals=self.decimals_setting)
        second_arr = np.around((self.arr - self.x_mean) / self.x_max,
decimals=self.decimals_setting)
        print("均值归一法：\n 公式一(分母 max):{}\n 公式二(分母 max-
min):{}".format(first_arr, second_arr))

    def Vector(self):
        if self.arr.sum() != 0:
            arr = np.around((self.arr / self.arr.sum()),
decimals=self.decimals_setting)
        else:
            arr = "未满足使用 sum()的要求(数据的和不为 0)"
        print("向量归一法:{}".format(arr))

    def exponential(self):
        if all(x > 1 for x in self.arr):
            first_arr = np.around(np.log10(self.arr) / np.log10(self.x_max),
decimals=self.decimals_setting)
        else:
            first_arr = "未满足使用 log10 的要求(数据均大于 1)"
        second_arr = np.around(np.exp(self.arr) / sum(np.exp(self.arr)),
decimals=self.decimals_setting)
        third_arr = np.around(1 / (1 + np.exp(-1 * self.arr)),
decimals=self.decimals_setting)
        print("lg 函数:{}\nSoftmax 函数:{}\nSigmoid 函数:{}\n".format(first_arr,
second_arr, third_arr))


if __name__ == "__main__":
    original_data1 = [-2, -1, 0, 1, 2, 3, 4]
    original_data2 = [-3, -2, -1, 0, 1, 2, 3]
    # 创建类的实例，并传递数组作为参数
    data1 = DataNorm(original_data1)
    data2 = DataNorm(original_data2)

    def FuckingFunction(data):
        data.Min_MaxNorm()
        data.Z_ScoreNorm()
        data.Decimal_ScalingNorm()
        data.MeanNorm()
        data.Vector()
        data.exponential()

    FuckingFunction(data1)
    FuckingFunction(data2)
```

输出：

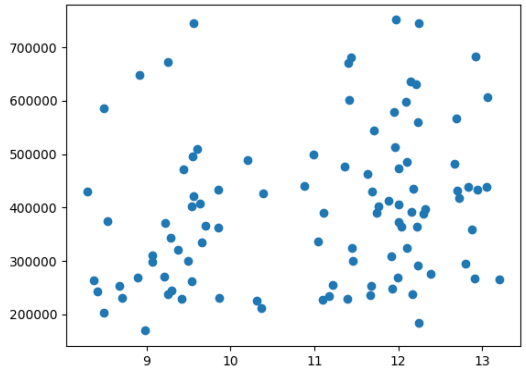
```
Min_Max标准化:[0.    0.167 0.333 0.5   0.667 0.833 1.   ]
Z_Score标准化:[-1.5 -1.  -0.5  0.   0.5  1.   1.5]
小数定标标准化:[-0.2 -0.1  0.   0.1  0.2  0.3  0.4]
均值归一法:
公式一(分母max):[-0.5   -0.333 -0.167  0.     0.167  0.333  0.5  ]
公式二(分母max-min):[-0.75 -0.5  -0.25  0.    0.25  0.5   0.75]
向量归一法:[-0.286 -0.143  0.     0.143  0.286  0.429  0.571]
lg函数:未满足使用log10的要求(数据均大于1)
Softmax函数:[0.002 0.004 0.012 0.032 0.086 0.233 0.633]
Sigmoid函数:[0.119 0.269 0.5   0.731 0.881 0.953 0.982]
```

```
Min_Max标准化:[0.    0.167 0.333 0.5   0.667 0.833 1.   ]
Z_Score标准化:[-1.5 -1.  -0.5  0.   0.5  1.   1.5]
小数定标标准化:[-0.3 -0.2 -0.1  0.   0.1  0.2  0.3]
均值归一法:
公式一(分母max):[-0.5   -0.333 -0.167  0.     0.167  0.333  0.5  ]
公式二(分母max-min):[-1.    -0.667 -0.333  0.     0.333  0.667  1.   ]
向量归一法:未满足使用sum()的要求(数据的和不为0)
lg函数:未满足使用log10的要求(数据均大于1)
Softmax函数:[0.002 0.004 0.012 0.032 0.086 0.233 0.633]
Sigmoid函数:[0.047 0.119 0.269 0.5   0.731 0.881 0.953]
```

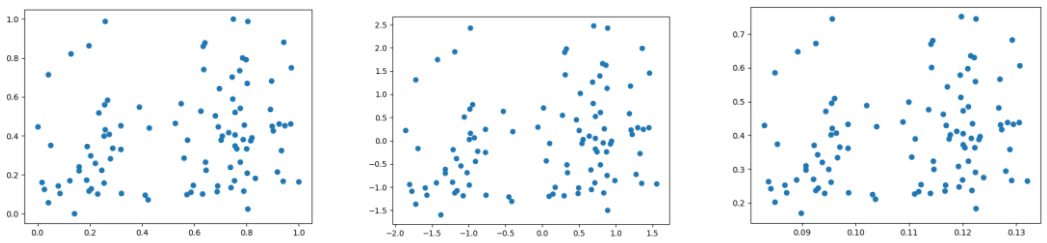
实际应用：

```python
import numpy as np
import pandas as pd
import sklearn.preprocessing as skp
import matplotlib.pyplot as plt

df = pd.read_csv('input/Ch3-StockTrading.csv')
print(df.describe())

# 选择最后 100 行的 close 和 volume 两列，.values 将 DataFrame 转换为 NumPy 数组
data_original = df.tail(100)[['close', 'volume']].values
plt.scatter(data_original[:, 0], data_original[:, 1])
plt.show()
```

	Unnamed: 0	open	close	high	low	volume	code
count	640.000000	640.000000	640.000000	640.000000	640.000000	6.400000e+02	640.0
mean	319.500000	11.990019	11.991128	12.236808	11.756233	2.269220e+05	300274.0
std	184.896367	3.682276	3.688834	3.791381	3.574300	1.714809e+05	0.0
min	0.000000	5.220000	5.330000	5.470000	5.170000	3.066200e+04	300274.0
25%	159.750000	9.568500	9.582500	9.717500	9.402250	9.588225e+04	300274.0
50%	319.500000	11.275000	11.285000	11.439500	11.137000	1.879755e+05	300274.0
75%	479.250000	15.060500	14.976500	15.373500	14.726000	3.026530e+05	300274.0
max	639.000000	21.870000	21.563000	22.495000	20.749000	1.472658e+06	300274.0




```python
data_scale = skp.MinMaxScaler().fit_transform(data_original)
plt.scatter(data_scale[:, 0], data_scale[:, 1])
plt.show()

data_zscore = skp.scale(data_original)
plt.scatter(data_zscore[:, 0], data_zscore[:, 1])
plt.show()

scaling_factor_close = 10**np.ceil(np.log10(np.max(np.abs(data_original[:, 0]))))
scaling_factor_volume = 10**np.ceil(np.log10(np.max(np.abs(data_original[:, 1]))))
data_decimal = data_original
for index in range(len(data_decimal)):
    data_decimal[index, 0] = data_decimal[index, 0]/scaling_factor_close
    data_decimal[index, 1] = data_decimal[index, 1]/scaling_factor_volume
plt.scatter(data_decimal[:, 0], data_decimal[:, 1])
plt.show()
```



One-hot 编码

```
from sklearn import preprocessing

enc = preprocessing.OneHotEncoder()  # 创建对象
enc.fit([[0, 0, 3], [1, 1, 0], [0, 2, 1], [1, 0, 2]])  # 拟合
array = enc.transform([[0, 1, 3]]).toarray()  # 转化
print(array)
```

输出

```
[[1. 0. 0. 1. 0. 0. 0. 0. 1.]]
```

fit 的具体过程：

对于训练对象 $\begin{pmatrix}0&0&3\\1&1&0\\0&2&1\\1&0&2\end{pmatrix}$：第一列是二元特征(0,1)，编码为 $\begin{matrix}0\to(1,0)\\1\to(0,1)\end{matrix}$。第二列是三元特征(0,1,2)，编码为 $\begin{matrix}0\to(1,0,0)\\1\to(0,1,0)\\2\to(0,0,1)\end{matrix}$。第三列是四元特征(0,1,2,3)，编码为 $\begin{matrix}0\to(1,0,0,0)\\1\to(0,1,0,0)\\2\to(0,0,1,0)\\3\to(0,0,0,1)\end{matrix}$。

因此对于[0,1,3]，第一列是 0，对应(1,0)，第二列的 1 对应(0,1,0)，第三列的 3 对应(0,0,0,1)，所以输出的是 1,0,0,1,0,0,0,0,1。

TF-IDF

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer

tag_list = ['hutao klee keqing',
            'klee klee klee',
            'hutao klee klee',
            'klee hutao hutao']
vectorizer = CountVectorizer()  # 将文本中的词语转换为词频矩阵
X = vectorizer.fit_transform(tag_list)  # 计算每个词语出现的次数
print(X)  # X 形如(a,b) c，表示再第 a 个文档中编号为 b 的单词出现了 c 次
print(vectorizer.vocabulary_)  # word_dict。顺序对应出现顺序，编码依据字母顺序

transformer = TfidfTransformer()
tfidf = transformer.fit_transform(X)  # 将词频矩阵 X 统计成 TF-IDF 值
print(tfidf.toarray())
```

输出(左：X、vectorizer.vocabulary_，右：tfidf.toarray())

```
  (0, 0)    1
  (0, 2)    1
  (0, 1)    1
  (1, 2)    3
  (2, 0)    1
  (2, 2)    2
  (3, 0)    2
  (3, 2)    1
{'hutao': 0, 'klee': 2, 'keqing': 1}
```

```
[[0.49248889 0.77157901 0.40264194]
 [0.         0.         1.        ]
 [0.52173612 0.         0.85310692]
 [0.92564688 0.         0.37838849]]
```

离散化

```
import pandas as pd
from sklearn.cluster import KMeans

df_train = pd.read_csv('input/Ch3-E-Commerce.csv', encoding='ISO-8859-1')
total = df_train.isnull().sum().sort_values(ascending=False)  # 计算了每列中的缺失值总数，并按降序排列
```

```
percent = (df_train.isnull().sum() /
df_train.isnull().count()).sort_values(ascending=False)  # 缺失值百分比 降序排列
missing_data = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])  # 合并 total 与 percent
print(missing_data)
price = df_train['UnitPrice']  # 通过直方图可以发现其分布极其不均衡(横跨范围很大[-11062.06, 38970.0]，但非常集中[10%:0.63, 90%:7.95])

""" 等宽离散化 """
K = 5  # 将价格划分为 K 个区间
df_train['price_discretized_1'] = pd.cut(price, K, labels=range(K))  # pd.cut 将 price 划分为 K 个区间，labels 即组名称
print(df_train.groupby('price_discretized_1')['price_discretized_1'].count())  # 照 price_discretized_1 列的值进行分组，并计算每个组的数量，最后打印结果

""" 等频离散化 """
w = [1.0*i/K for i in range(K+1)]  # [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]
# 为什么是加 2？首先，分割点应该比区间 K 多 1，其次 describe 自带一个 50%项(下文会删除这个 50%项)。
# 如果只+1 会发现 w 少了"100%"这一项，需要 pd.concat([w, pd.Series([price.max()], index=["100%"])])重新补回来
w = price.describe(percentiles=w)[4:4+K+2]  # 前四项是 count,mean,std,min
w = w.drop('50%')  # 删除 index="50%"的 w
df_train['price_discretized_2'] = pd.cut(price, w, labels=range(K))  # 原数据集 541909 项，等频后 541907 项
print(df_train.groupby('price_discretized_2')['price_discretized_2'].count())

""" K 均值离散化 """
price_re = price.values.reshape((price.index.size, 1))  # price.index.size = 541909。将 p 一维数组 rice 重塑为二维[541909,1]
k_model = KMeans(n_clusters=K, n_init=10)
df_train['price_discretized3'] = k_model.fit_predict(price_re)
print(df_train.groupby('price_discretized3')['price_discretized3'].count())
```

输出(省略查看缺失值的输出)

```
price_discretized_1
0          2
1     541897
2          9
3          0
4          1
Name: price_discretized_1, dtype: int64
price_discretized_2
0     113124
1     119971
2      93042
3     123061
4      92709
Name: price_discretized_2, dtype: int64
price_discretized3
0     541861
1          1
2          9
3         36
4          2
Name: price_discretized3, dtype: int64
```

6.Sklearn

数据集的使用

```
from sklearn import datasets
print(dir(datasets))  # 这个列表包括了 datasets 模块中的各种函数、类、子模块等。
```

输出

```
['__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__getattr__', '__loader__', '__name__', '__package__', '__path__', '__spec__', '_arff_parser', '_base', '_california_housing', '_covtype', '_kddcup99', '_lfw', '_olivetti_faces', '_openml', '_rcv1', '_samples_generator', '_species_distributions', '_svmlight_format_fast', '_svmlight_format_io', '_twenty_newsgroups', 'clear_data_home', 'dump_svmlight_file', 'fetch_20newsgroups', 'fetch_20newsgroups_vectorized', 'fetch_california_housing', 'fetch_covtype', 'fetch_kddcup99', 'fetch_lfw_pairs', 'fetch_lfw_people', 'fetch_olivetti_faces', 'fetch_openml', 'fetch_rcv1', 'fetch_species_distributions', 'get_data_home', 'load_breast_cancer', 'load_diabetes', 'load_digits', 'load_files', 'load_iris', 'load_linnerud', 'load_sample_image', 'load_sample_images', 'load_svmlight_file', 'load_svmlight_files', 'load_wine', 'make_biclusters', 'make_blobs', 'make_checkerboard', 'make_circles', 'make_classification', 'make_friedman1', 'make_friedman2', 'make_friedman3', 'make_gaussian_quantiles', 'make_hastie_10_2', 'make_low_rank_matrix', 'make_moons', 'make_multilabel_classification', 'make_regression', 'make_s_curve', 'make_sparse_coded_signal', 'make_sparse_spd_matrix', 'make_sparse_uncorrelated', 'make_spd_matrix', 'make_swiss_roll', 'textwrap']
```

①数据集：Iris

```
Iris = datasets.load_iris()  # 导入鸢尾花数据集
data = Iris.data  # 获得样本特征向量
target = Iris.target  # 获得样本 label
# print(data, target)  # 查看 data(4 个属性)与 target(3 个属性值)
print(type(data), type(target))  # 均为<class 'numpy.ndarray'>
print(Iris.feature_names)  # 4 个属性特征
print(Iris.target_names)  # 3 种鸢尾花的名称
print(Iris.data.shape, Iris.target.shape)  # 规格
```

导入： Iris = datasets.load_iris()

描述： 含了三种不同种类的鸢尾花(Iris)的测量数据。

目的： 用于模式识别、分类问题、聚类分析和数据可视化等领域。

规格： data(150, 4)　target(150,)

属性解释：

data

属性	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
解释	花萼长度	花萼宽度	花瓣长度	花瓣宽度
第一行的值	5.1	3.5	1.4	0.2

target

setosa	versicolor	virginica
山鸢尾	变色鸢尾	维吉尼亚鸢尾
0	1	2

注：

山鸢尾 Iris setosa：一种较小的鸢尾花，通常花瓣较短，直立生长，花朵白色。

变色鸢尾 Iris versicolor：具有较长的花瓣，通常为淡蓝色或紫色，花朵的颜色会变化。

维吉尼亚鸢尾 Iris virginica：花瓣较长，通常为深紫色。

②数据集 Boston

```
import numpy as np
import pandas as pd
data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep=r"\s+", skiprows=22, header=None)
data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
target = raw_df.values[1::2, 2]
print(data.shape, target.shape)  # 规格
np.set_printoptions(threshold=np.inf)  # 设置打印选项，将数组的所有元素都显示出来
np.set_printoptions(suppress=True)  # 设置打印选项，将数组的元素以普通小数表示法显示
print(data)
print(target)
```

描述： 提供了与波士顿地区不同社区相关的一些特征，以及这些社区的房价中位数。

目的：房价预测。评估各种回归算法，如线性回归、决策树回归、随机森林回归等，以了解不同特征如何影响房价，以及如何建立准确的房价预测模型。

规格：data(506, 13)　target(506,)

属性解释：

属性	解释	第一行的值
CRIM	犯罪率	0.00632
ZN	>25000ft^2 的规划住宅用地面积	18.00
INDUS	非零售商业用地占比	2.310
CHAS	是否临 Charles 河(1 临 0 不临)	0
NOX	一氧化氮浓度	0.5380
RM	住宅平均房间数目	6.5750
AGE	1940 年以前建成用房的居住率	65.20
DIS	和波士顿五大就业中心的距离	4.0900
RAD	是否容易上高速路	1
TAX	全额财产税的税率	296.0
PTRATIO	城镇小学教师的比例	15.30
B	黑人所占人口比例	396.9
LSTAT	低收入人群比例	4.98
MEDV	居住房屋的房价中位数(千美元)	24.00

③数据集 Digits

```
import matplotlib.pyplot as plt
Digits = datasets.load_digits()  # 导入手写数字数据集
data = Digits.data
target = Digits.target
images = Digits.images
print(data.shape)  # (1797, 64)
print(target.shape)  # (1797,)
print(images.shape)  # (1797, 8, 8)
print(Digits.feature_names)  # 64 个属性特征 pixel_0_0~pixel_7_7
print(Digits.target_names)  # 数字 0~9
plt.matshow(Digits.images[0])  # 绘制第一个 image
plt.show()
print(Digits.target[0])  # 查看第一个 image 对应是数字是几
```

导入：Digits = datasets.load_digits()

描述：包含一系列手写数字的图像，每个图像都代表一个从 0 到 9 的数字。

目的：数字识别和分类任务(支持向量机、神经网络、决策树等)。

规格：data (1797, 64)　target(1797,)　image(1797, 8, 8)

数据特点：总共 1797 张 8x8 像素的图像。每个图像代表一个手写数字，10 个类别分别对应一个数字。每个像素的值表示图像中的灰度强度。

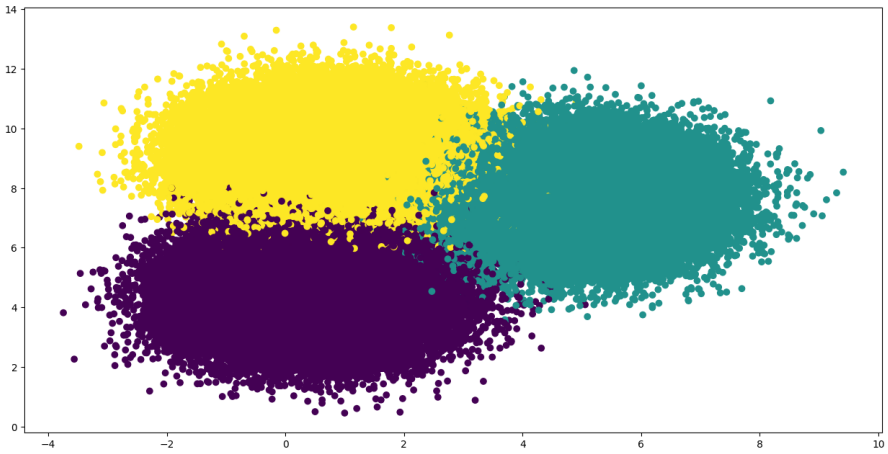
注：

data 与 image 的区别：data 被展平为一维数组，image 以 8*8 二维矩阵保存。data 可以被看作是将 images 中的每个二维图像展平为一个一维数组的结果(扁平化结果)。

④自拟数据集

```
from sklearn.datasets import make_regression
from sklearn.datasets import make_classification
from sklearn.datasets import make_blobs
```

```
X, y = make_regression(n_samples=300, n_features=10, n_targets=1)  # 创建回归数据集
X, y = make_classification(n_samples=300, n_features=10, n_classes=2)  # 创建分类数据集
X, y = make_blobs(n_samples=221027, n_features=2, centers=3)  # 创建分类数据集
plt.scatter(X[:, 0], X[:, 1], c=y)  # 展示聚类的散点图
plt.show()
```



make_regression:

n_samples:样本数量。n_features:特征数量。n_targets:生成的目标(输出)的数量。

make_classification:

n_samples:样本数量。n_features:特征数量。n_classes:类别的数量。

make_blobs:

n_samples:样本数量。n_features:特征数量。centers:聚类的中心数量(簇的数量)。

数据集的拆分

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(data, target, test_size=0.4, random_state=0)
print(X_train.shape, X_test.shape)  # (90, 4) (60, 4)  [y_test,y_train 是(90,) (60,)]
```

train_test_split:

data：特征向量。target：样本标签。

test_size：测试集大小，浮点数表示测试样本所占百分比，整数表示多少个测试样本。

random_state：随机种子，非 0 代表随机数编号，每次运行代码时，分割数据的结果都将是相同的。0 或不填每次产生的划分都是不一样的。

返回四个数组：X_train：训练集的特征向量。X_test：测试集的特征向量。y_train：训练集的标签。y_test：测试集的标签。

预处理模块、常用模型的导入语句

```
from sklearn.preprocessing import StandardScaler  # 数据标准化
from sklearn.preprocessing import MinMaxScaler  # 数据归一化
from sklearn.preprocessing import LabelEncoder  # 标签编码
from sklearn.preprocessing import OneHotEncoder  # 独热编码
from sklearn.impute import SimpleImputer  # 缺失值处理
from sklearn.preprocessing import PolynomialFeatures  # 多项式特征生成
from sklearn.feature_selection import SelectKBest  # 特征选择
from sklearn.feature_selection import VarianceThreshold  # 方差阈值
from sklearn.decomposition import PCA  # 主成分分析
from sklearn.decomposition import NMF  # 非负矩阵分解
```

```
from sklearn.linear_model import LinearRegression  # 线性回归
from sklearn.linear_model import LogisticRegression  # 逻辑回归
from sklearn.svm import SVC  # 支持向量机分类
from sklearn.svm import SVR  # 支持向量机回归
from sklearn import naive_bayes  # 朴素贝叶斯算法 NB
from sklearn import tree  # 决策树算法
from sklearn import neighbors  # K 临近算法
from sklearn import neural_network  # 神经网络
from sklearn.ensemble import RandomForestClassifier  # 随机森林分类
from sklearn.ensemble import RandomForestRegressor  # 随机森林回归
from sklearn.neighbors import KNeighborsClassifier  # K 近邻分类
from sklearn.neighbors import KNeighborsRegressor  # K 近邻回归
from sklearn.tree import DecisionTreeClassifier  # 决策树分类
from sklearn.tree import DecisionTreeRegressor  # 决策树回归
from sklearn.cluster import KMeans  # K 均值聚类
from sklearn.naive_bayes import GaussianNB  # 朴素贝叶斯分类
from sklearn.neural_network import MLPClassifier  # 多层感知器分类
from sklearn.neural_network import MLPRegressor  # 多层感知器回归
from sklearn.ensemble import GradientBoostingClassifier  # 梯度提升分类
from sklearn.ensemble import GradientBoostingRegressor  # 梯度提升回归
```

模型预测与评估

依据 Iris 数据集的导入与模型的导入，有：

```
model = SVC()  # 创建一个 SVC 模型的实例
model.fit(X_train, y_train)  # 拟合模型--使用训练数据进行模型训练
y_pre = model.predict(X_test)  # 预测模型--使用训练好的模型对测试数据进行预测
print(model.get_params())  # 获得模型的参数
print(model.score(X_test, y_test))  # 模型在测试数据的准确度
print(y_pre)  # 模型对测试数据的预测结果
print(y_test)  # 测试数据的实际结果
```

输出(预测与实际不相同的 4 个数据已经加粗)

```
{'C': 1.0, 'break_ties': False, 'cache_size': 200, 'class_weight': None, 'coef0': 0.0, 'decision_function_shape': 'ovr', 'degree': 3, 'gamma':
'scale', 'kernel': 'rbf', 'max_iter': -1, 'probability': False, 'random_state': None, 'shrinking': True, 'tol': 0.001, 'verbose': False}
0.9333333333333333
[2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 1 0 0 1 1 0 2 1 0 2 2 1 0
 2 1 1 2 0 2 0 0 1 2 2 1 2 1 2 1 1 2 2 1 2 1 2]
[2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0
 1 1 1 2 0 2 0 0 1 2 2 2 2 1 2 1 1 2 2 2 2 1 2]
```

Sklearn 的模型接口都是类似的。

fit()用于模型的训练，传入两个参数(训练集数据和训练集标签)。

predict()用于模型的预测，传入测试集数据，输出预测的结果。

get_pramas()用于获得模型参数。

score()用于对训练好的模型打分。

模型保存与加载

两种方法二选一：

```
import pickle
with open('model.pickle', 'wb') as f:  # 模型保存
    pickle.dump(model, f)
with open('model.pickle', 'rb') as f:  # 模型加载
    model = pickle.load(f)
model.predict(X_test)

import joblib
joblib.dump(model, 'model2.pickle')  # 模型保存
model = joblib.load('model2.pickle')  # 模型加载
```

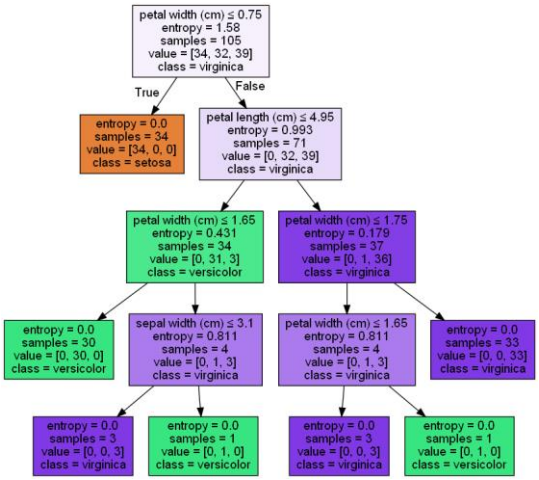
7.分类

决策树

```
from sklearn import datasets
from sklearn import tree
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.tree import export_graphviz
from six import StringIO
import pydotplus

Iris = datasets.load_iris()  # 导入鸢尾花数据集
X = Iris.data  # 获得样本特征向量
Y = Iris.target  # 获得样本 label
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,
random_state=0)
print("target 的名称:%s" % Iris.target_names)

tree = tree.DecisionTreeClassifier(criterion='entropy')  # 使用的划分标准是信息熵
entropy
tree.fit(x_train, y_train)
print("决策树模型的训练集准确率:%.3f" % tree.score(x_train, y_train))
print("决策树模型的测试集准确率:%.3f" % tree.score(x_test, y_test))
y_hat = tree.predict(x_test)
print(classification_report(y_test, y_hat, target_names=Iris.target_names))

# 生成决策树可视化的 DOT 文件
dot_data = StringIO()
export_graphviz(tree, out_file=dot_data, filled=True,
feature_names=Iris.feature_names, class_names=Iris.target_names,
                fontname='Arial', special_characters=True)
dot_data = dot_data.getvalue()
dot_data = dot_data.replace('\n', '')  # 去除右上角黑块
# 使用 pydotplus 生成图像
graph = pydotplus.graph_from_dot_data(dot_data)
graph.write_png("output/iris_decision_tree.png")  # 将决策树保存为 PNG 文件
```

输出

```
target 的名称:['setosa' 'versicolor' 'virginica']
决策树模型的训练集准确率:1.000
决策树模型的测试集准确率:0.978
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        16
  versicolor       1.00      0.94      0.97        18
   virginica       0.92      1.00      0.96        11

    accuracy                           0.98        45
   macro avg       0.97      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45
```



在训练集上准确率为 100%，表示模型能完美地对训练数据分类，但也暗示可能过拟合。

在测试集上准确率为 97.8%。表明模型在未见过的数据上表现很好，具有较高泛化能力。

第一个矩阵代表三个 target 的**查准率、查全率、F1 度量、支持的样本数量**。

第二个矩阵 Accuracy **精度**：整个模型在所有类别上的正确分类比例。

Macro average **宏平均**：对每个类别的评估指标取平均值，不考虑类别的样本数量，以 precision 为例，$\frac{1.00+1.00+0.92}{3}=0.97333$。

Weighted average **加权平均**：对每个类别的评估指标取加权平均值，考虑不同类别的样本数量。以 precision 为例，$1.00\times\frac{16}{45}+1.00\times\frac{18}{45}+0.92\times\frac{11}{45}=0.98044$。

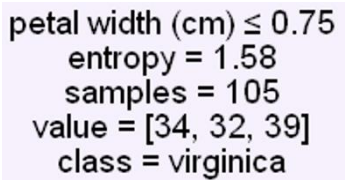
决策树图形的参数讲解：

第一行：划分依据。**第二行**：熵。

第三行：本次划分前有 105 个样本。

第四行：样本标签有三个，数量依次为 34,32,39。

第五行：代表类别(叶子结点的 class 才代表最终标签)。



KNN

距离度量-欧氏距离的计算

```
import numpy as np

vector1 = np.mat([1, 2, 3])  # [[1 2 3]]
vector2 = np.mat([4, 5, 6])  # [[4 5 6]]
distance = np.sqrt((vector1-vector2) * (vector1-vector2).T)  # [[-3 -3 -3]] * [[-3]
[-3] [-3]] = [[5.19615242]]
print(distance)
```

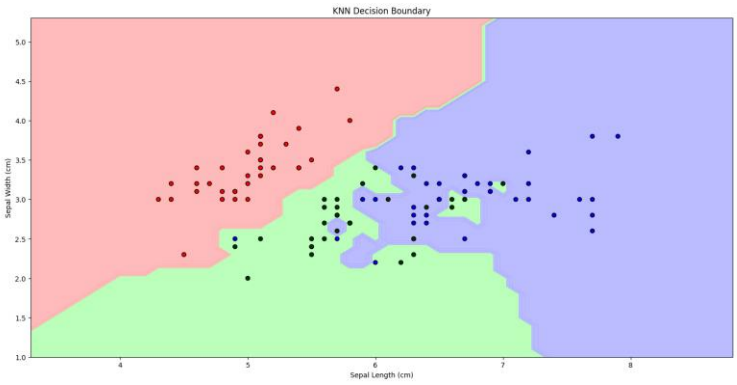
KNN-模型拟合、样本预测、二维绘图

```
from sklearn import datasets
from sklearn import neighbors
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

Iris = datasets.load_iris()  # 导入鸢尾花数据集
X = Iris.data  # 获得样本特征向量
Y = Iris.target  # 获得样本 label
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,
random_state=0)
print("target 的名称:%s" % Iris.target_names)

KNN = neighbors.KNeighborsClassifier(n_neighbors=3)
KNN.fit(x_train, y_train)
print("KNN 模型的训练集准确率:%.3f" % KNN.score(x_train, y_train))
print("KNN 模型的测试集准确率:%.3f" % KNN.score(x_test, y_test))
y_hat = KNN.predict(x_test)
print(classification_report(y_test, y_hat, target_names=Iris.target_names))

new_sample = np.array([5.1, 3.5, 1.4, 0.2]).reshape(1, -1)  # 新样本的特征值(其实是
Iris 数据集的第一个数据)
predicted_class = KNN.predict(new_sample)  # 运行后得知 predicted_class 的值为[0]，因此
下一行代码里的 predicted_class[0]就是 0。
predicted_class_name = Iris.target_names[predicted_class[0]]
print("样本[5.1, 3.5, 1.4, 0.2]预测的类别是:", predicted_class_name)

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])  # 给不同区域赋以颜色
cmap_bold = ListedColormap(['#FF0000', '#003300', '#0000FF'])  # 给不同属性的点赋以颜
色
# 定义绘图边界
feature1_min, feature1_max = x_train[:, 0].min() - 1, x_train[:, 0].max() + 1  # 第
一个属性 sepal length 花萼长度
feature2_min, feature2_max = x_train[:, 1].min() - 1, x_train[:, 1].max() + 1  # 第
二个属性 sepal width 花萼宽度
```

```
xx, yy = np.meshgrid(np.arange(feature1_min, feature1_max, 0.1),
np.arange(feature2_min, feature2_max, 0.1))  # 生成网格点
KNN = neighbors.KNeighborsClassifier(n_neighbors=3)
KNN.fit(x_train[:, :2], y_train)  # 以二维重新预测，属性变为原先训练集的一二个属性，
target 不变
print("二维 KNN 模型的训练集准确率:%.3f" % KNN.score(x_train[:, :2], y_train))
print("二维 KNN 模型的测试集准确率:%.3f" % KNN.score(x_test[:, :2], y_test))
# 将预测的结果在平面坐标中画出其类别区域
Z = KNN.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
# 绘制决策边界和训练样本
plt.contourf(xx, yy, Z, cmap=cmap_light, alpha=0.8)
plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train, marker='o', edgecolors='k',
cmap=cmap_bold)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.title('KNN Decision Boundary')
plt.show()
```

输出

```
target 的名称:['setosa' 'versicolor' 'virginica']
KNN 模型的训练集准确率:0.962
KNN 模型的测试集准确率:0.978
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        16
  versicolor       1.00      0.94      0.97        18
   virginica       0.92      1.00      0.96        11

    accuracy                           0.98        45
   macro avg       0.97      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45

样本[5.1, 3.5, 1.4, 0.2]预测的类别是: setosa
二维 KNN 模型的训练集准确率:0.857
二维 KNN 模型的测试集准确率:0.733
```



试验可知 n_neighbor 取 5 的时候比 3 在 train 上好一些。

绘制 KNN 边界的时候有一个问题，边界应当是连续的，但数据集是非常离散的。因此需要使用一个间隔(这里是 0.1，也可以取得更小，但运行时间会显著上升)来近似连续。但是对于不存在于数据集的点，如何我们选定的二维空间对应到四维空间里呢？或者说，对于给定的某个点(x,y)，已知的只有花萼长宽，没有花瓣长宽，缺失的数据没办法填补。并且，花瓣长宽不完全取决于花萼长宽，因此可能相同的花萼长宽对应不同的花瓣长宽时，会使得花的 target 发生变化。所以在二维图像里没办法给出全部信息，使得 KNN 只能重新拟合二维，因为剩下两维没办法确定。

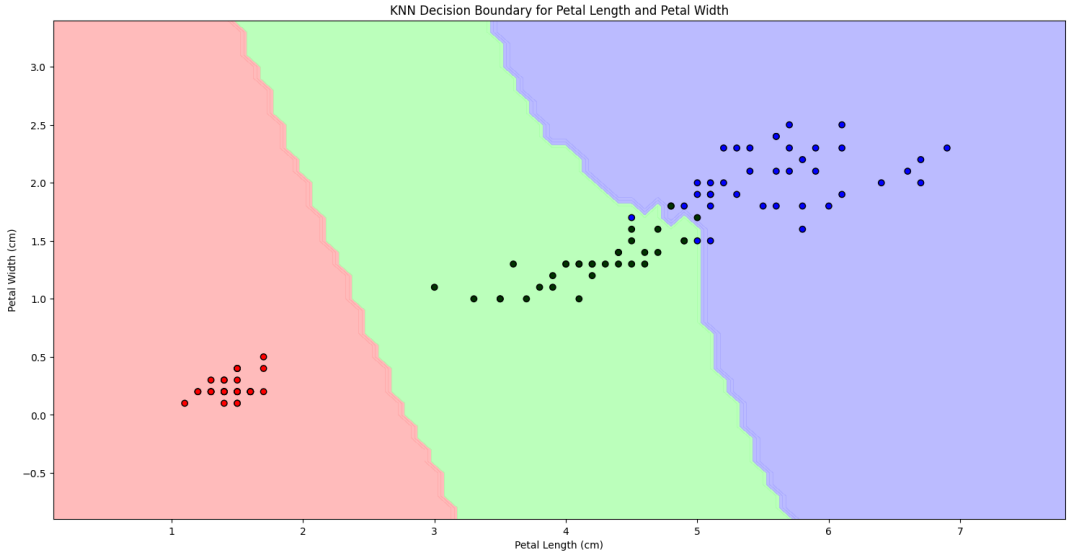
可以看到使用花萼属性的拟合其实不好，但使用花瓣拟合就很不错（可以预先查看散点图，就能发现花瓣很明显有更好的分类效果）：

```
feature3_min, feature3_max = x_train[:, 2].min() - 1, x_train[:, 2].max() + 1  # 第
三个属性 petal length 花瓣长度
feature4_min, feature4_max = x_train[:, 3].min() - 1, x_train[:, 3].max() + 1  # 第
四个属性 petal width 花瓣宽度
xx, yy = np.meshgrid(np.arange(feature3_min, feature3_max, 0.1),
np.arange(feature4_min, feature4_max, 0.1))
KNN = neighbors.KNeighborsClassifier(n_neighbors=3)
KNN.fit(x_train[:, 2:4], y_train)
print("后两个属性的 KNN 模型的训练集准确率:%.3f" % KNN.score(x_train[:, 2:4], y_train))
print("后两个属性的 KNN 模型的测试集准确率:%.3f" % KNN.score(x_test[:, 2:4], y_test))
Z = KNN.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cmap_light, alpha=0.8)
plt.scatter(x_train[:, 2], x_train[:, 3], c=y_train, marker='o', edgecolors='k',
```

```
cmap=cmap_bold)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.xlabel('Petal Length (cm)')
plt.ylabel('Petal Width (cm)')
plt.title('KNN Decision Boundary for Petal Length and Petal Width')
plt.show()
```

输出

```
后两个属性的 KNN 模型的训练集准确率:0.962
后两个属性的 KNN 模型的测试集准确率:0.978
```



另外，KNN 要求所有的属性必须经过标准化处理，以免属性之间的数量级相差过大而导致偏差。KNN 可以自动忽略缺失值。对异常数据不敏感，具有较好的抗噪性。

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()  # 创建一个 Z-Score 标准化处理器
x_train_scaled = scaler.fit_transform(x_train)  # 对训练数据进行标准化处理
x_test_scaled = scaler.transform(x_test)  # 对测试数据进行相同的标准化处理
```

其余与之前的基本一致。

朴素贝叶斯

```
from sklearn import datasets
from sklearn import naive_bayes
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split

Iris = datasets.load_iris()
X = Iris.data
Y = Iris.target
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,
random_state=0)

bayes = naive_bayes.GaussianNB()
bayes.fit(x_train, y_train)
print("贝叶斯模型训练集的准确率:%.3f" % bayes.score(x_train, y_train))
print("贝叶斯模型测试集的准确率:%.3f" % bayes.score(x_test, y_test))
y_hat = bayes.predict(x_test)
print(classification_report(y_test, y_hat, target_names=Iris.target_names))
```

输出

```
贝叶斯模型训练集的准确率:0.943
贝叶斯模型测试集的准确率:1.000
              precision    recall  f1-score   support
```

setosa	1.00	1.00	1.00	16
versicolor	1.00	1.00	1.00	18
virginica	1.00	1.00	1.00	11
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

SVM

```
from sklearn import datasets
from sklearn import neighbors
from sklearn import svm
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split

Iris = datasets.load_iris()
X = Iris.data
Y = Iris.target
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,
random_state=0)

SVM = svm.SVC()  # Support Vector Classification 是用于分类问题的 SVM 实现
SVM.fit(x_train, y_train)
print("SVM 模型训练集的准确率:%.3f" % SVM.score(x_train, y_train))
print("SVM 模型测试集的准确率:%.3f" % SVM.score(x_test, y_test))
y_hat = SVM.predict(x_test)
print(classification_report(y_test, y_hat, target_names=Iris.target_names))
```

输出

```
SVM 模型训练集的准确率:0.971
SVM 模型测试集的准确率:0.978
```

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	16
versicolor	1.00	0.94	0.97	18
virginica	0.92	1.00	0.96	11
accuracy			0.98	45
macro avg	0.97	0.98	0.98	45
weighted avg	0.98	0.98	0.98	45

8.回归

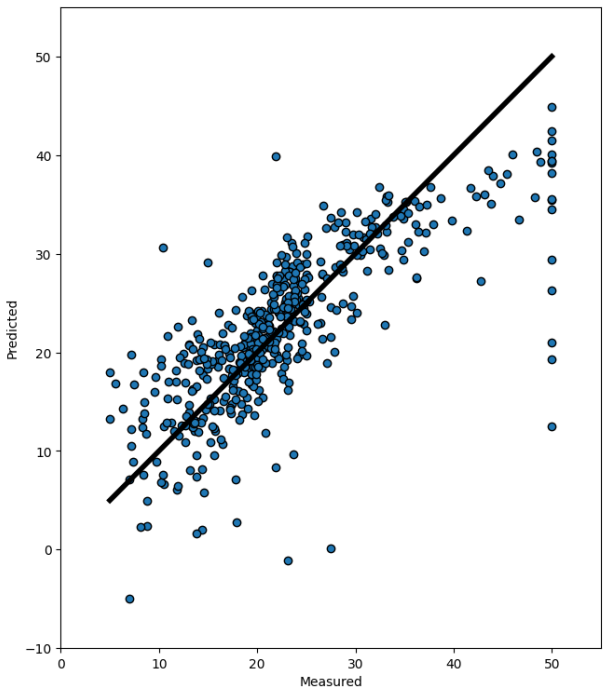
线性回归

```
from sklearn.model_selection import cross_val_predict
from sklearn import linear_model
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep=r"\s+", skiprows=22, header=None)
data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])  # 属性
target = raw_df.values[1::2, 2]  # 预测值
print(data.shape, target.shape)  # 规格(506, 13) (506,)

lr = linear_model.LinearRegression()  # 线性回归
# 使用交叉验证。将数据分成 10 折(由 cv=10 指定)，每次使用 9 折数据训练模型，然后用模型预测第 10 折数据，并将预测结果保存下来。重复 10 次。
predicted = cross_val_predict(lr, data, target, cv=10)

# 查看 target 的最小和最大值以便绘图
print("Target - Min:", min(target))  # 5.0
print("Target - Max:", max(target))  # 50.0
# 查看 predicted 的最小和最大值
print("Predicted - Min:", min(predicted))  # -4.9536551238824345
print("Predicted - Max:", max(predicted))  # 44.912287884174994

fig, ax = plt.subplots()
ax.scatter(target, predicted, edgecolors=(0, 0, 0))
# 绘制线条（从(5,5)到(50,50)）。这条直线表示理想情况下，如果模型的预测完全准确，所有的点都应该落在这条直线上。
ax.plot([target.min(), target.max()], [target.min(), target.max()], 'k-', lw=4)  # k:黑色，-:实线，lw:line width。
ax.set_xlim(0, 55)  # 设置 x 轴范围
ax.set_ylim(-10, 55)  # 设置 y 轴范围
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
plt.show()  # 不要问我为什么 target 有那么多 50 的数据点，我也很奇怪啊
```

输出

```
(506, 13) (506,)
Target - Min: 5.0
Target - Max: 50.0
Predicted - Min: -4.9536551238824345
Predicted - Max: 44.912287884174994
```

右图，横轴是准确值，纵轴是预测值。直线代表预测准确的点。预测值在直线之上/下代表预测过高/低。



线性回归的三种梯度下降方法

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

np.random.seed(42)  # 设置固定种子便于实验


""" ---线性回归--- """
""" 生成数据集 & 绘图 """
X = 2 * np.random.rand(727, 1)  # 727 个[0, 1)之间均匀分布的随机数 * 2
y = 4 + 3 * X + np.random.randn(727, 1)  # 4 + 3x + 标准正态分布误差项
plt.plot(X, y, 'b.')
plt.axis([0, 2, 1, 14])
plt.show()

""" 直接使用公式计算 & 绘图 """
X_b = np.c_[np.ones((727, 1)), X]  # 拼接数据，形如[[1.         0.32950847] [1.         0.62878135]...[1.
0.53402584]]
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)  # (X^T * X)^(-1) * X^T * y
print("公式计算的 θ: ", theta_best)  # [[4.02650858] [2.96474178]]很接近我们希望的系数 4 和 3
X_HeadAndTail = np.array([[0], [2]])  # [[0] [2]]
X_HeadAndTail_b = np.c_[np.ones((2, 1)), X_HeadAndTail]  # [[1. 0.] [1. 2.]]
y_predicted = X_HeadAndTail_b.dot(theta_best)  # [[4.02650858] [9.95599214]]
plt.plot(X_HeadAndTail, y_predicted, 'r--')
plt.plot(X, y, 'b.')
plt.axis([0, 2, 1, 14])
plt.show()

""" LinearRegression """
lin_reg = LinearRegression()
lin_reg.fit(X, y)
print("偏置参数: ", lin_reg.intercept_)  # 偏置参数 [4.02650858]
print("权重参数: ", lin_reg.coef_)  # 权重参数 [[2.96474178]]

""" 使用 MSE 的批量梯度下降 """
eta = 0.01  # 学习率 learning rate
n_iterations = 10000  # 迭代次数
n = len(X_b)  # 样本数量
theta = np.random.randn(2, 1)  # 形状为(2,1)的数组
for iteration in range(n_iterations):
    gradients = 2 / n * X_b.T.dot(X_b.dot(theta) - y)  # 梯度(使用均方误差损失函数 MSE)
    theta = theta - eta * gradients  # 参数更新
print("MSE 批量梯度下降迭代 10000 次的 θ: ", theta)

""" 三种梯度下降的对比"""
# 批量梯度下降 BGD(学习率对结果的影响)
theta_path_bgd = []  # bgd：Batch Gradient Descent 批量梯度下降
def plot_gradient_descent(theta, eta, store_bgd = None):
    n = len(X_b)
    plt.plot(X, y, 'b.')
    n_iterations = 200  # 迭代次数
    for iteration in range(n_iterations):
        gradients = 2 / n * X_b.T.dot(X_b.dot(theta) - y)  # 梯度(使用均方误差损失函数 MSE)
        theta = theta - eta * gradients  # 参数更新
        if store_bgd is not None:  # 因为 BGD 有三组对比实验，我们只需要存入一个 bgd 的值与后续 sgd,mgd 比较
            store_bgd.append(theta)  # 通过传入 store_bgd=theta_path_bgd 实现向 theta_path_bgd 里传值
        y_predicted = X_HeadAndTail_b.dot(theta)
        plt.plot(X_HeadAndTail, y_predicted, color=(0.1, 1, 0.5, min(8 * iteration / n_iterations, 0.2222 *
iteration / n_iterations + 0.7778)))  # color 的参数依次是红绿蓝和透明度 alpha，alpha 是分段函数
y=8*x(x<=0.1),y=0.2222x+0.7778(x>0.1)
        # print("x:", iteration / n_iterations, "\ty:", min(40*iteration/n_iterations,
0.204*iteration/n_iterations+0.796)) # 查看 color 中的函数对应的 x,y 值
        plt.text(X_HeadAndTail[-1], y_predicted[-1], f'{iteration}', color='red')  # 在当前迭代的最后一个点上添加文本标
签
    plt.xlabel('X')
    plt.ylabel('y')
    plt.axis([0, 2, 1, 14])
    plt.title('eta={}'.format(eta))
theta = np.random.randn(2, 1)
plt.figure(figsize=(10, 4))
plt.subplot(131)
plot_gradient_descent(theta, eta=0.01, store_bgd=theta_path_bgd)
plt.subplot(132)
plot_gradient_descent(theta, eta=0.1)
plt.subplot(133)
plot_gradient_descent(theta, eta=0.3)
plt.show()

# 随机梯度下降 SGD
theta_path_sgd = []
n = len(X_b)  # 数据个数
n_epochs = 50  # 迭代的轮数，即整个数据集被遍历的次数
t0 = 5
t1 = 50  # t0,t1 共同控制学习率的衰减
theta = np.random.randn(2, 1)
```

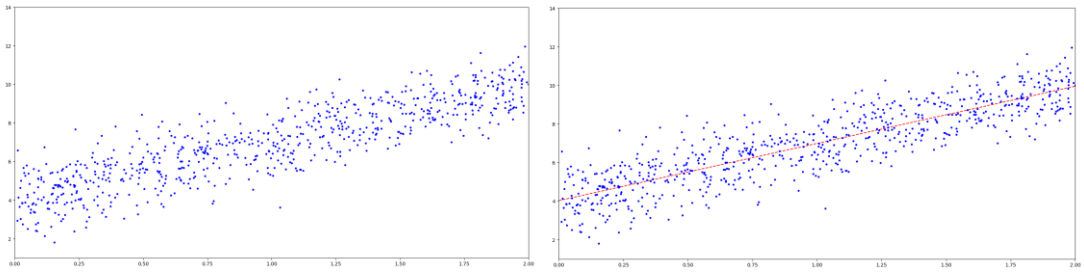
```
for epoch in range(n_epochs):
    for i in range(n):
        if epoch < 10 and i < n and i % 20 == 0:  # 绘制前十个 epoch 对应的拟合直线，每个 epoch 绘制间隔为 20 的直线(便于显示出明显的间隔)
            y_predicted = X_HeadAndTail_b.dot(theta)
            plt.plot(X_HeadAndTail, y_predicted, color=(0.1, 1, 0.5, i/n))
            if i == 0:  # 仅当每个 epoch 的第一个 i 的时候绘制出 epoch 的值
                plt.text(X_HeadAndTail[-1], y_predicted[-1], f'{epoch}', color='red')
        random_index = np.random.randint(n)  # 随机选择一个样本
        xi = X_b[random_index:random_index + 1]
        yi = y[random_index:random_index + 1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = t0 / (t1+(n_epochs*n+i))  # 当迭代次数(n_epochs*n+i)增加，学习率 eta 呈降低趋势
        theta = theta - eta * gradients
        theta_path_sgd.append(theta)
plt.plot(X, y, 'b.')
plt.axis([0, 2, 1, 14])
plt.show()

# 小批量梯度下降 Mini-batch Gradient Descent
theta_path_mgd = []
n_epochs = 800  # 为什么这里迭代次数远高于上面的才能收敛？因为每次 epoch 只迭代了 n/minibatch 次，虽然每次迭代有 minibatch 个数据样本点，但 minibatch 个数据样本点真的比 1 个数据样本点强很多吗？
minibatch = 16
theta = np.random.randn(2, 1)
for epoch in range(n_epochs):
    shuffled_index = np.random.permutation(n)  # 样本洗牌
    X_b_shuffled = X_b[shuffled_index]
    y_shuffled = y[shuffled_index]
    for i in range(0, n, minibatch):  # 从 0 开始，以步长 minibatch 迭代，直到 n-1 结束。
        if epoch < 160 and epoch % 10 == 0 and i % (20*minibatch) == 0:  # 只绘制前面的 epoch。对每个 epoch 依然只绘制部分迭代过程
            y_predicted = X_HeadAndTail_b.dot(theta)
            plt.plot(X_HeadAndTail, y_predicted, color=(0.1, 1, 0.5, i / n))
            if i == 0:  # 仅当每个 epoch 的第一个 i 的时候绘制出 epoch 的值
                plt.text(X_HeadAndTail[-1], y_predicted[-1], f'{epoch}', color='red')
        xi = X_b_shuffled[i: i+minibatch]
        yi = y_shuffled[i: i+minibatch]
        gradients = 2/minibatch * xi.T.dot(xi.dot(theta) - yi)
        eta = t0 / (t1 + (n_epochs*n/minibatch + i/minibatch))
        theta = theta - eta * gradients
        theta_path_mgd.append(theta)
plt.plot(X, y, 'b.')
plt.axis([0, 2, 1, 14])
plt.show()

# BGD,SGD,MGD 三者对比
theta_path_bgd = np.array(theta_path_bgd)
theta_path_sgd = np.array(theta_path_sgd)
theta_path_mgd = np.array(theta_path_mgd)
print("BGD:", theta_path_bgd[-1], "\nSGD:",  theta_path_sgd[-1], "\nMGD:", theta_path_mgd[-1])
plt.plot(theta_path_bgd[:, 0], theta_path_bgd[:, 1], 'b-o', label='BGD')
plt.plot(theta_path_sgd[:, 0], theta_path_sgd[:, 1], 'r-s', label='SGD')
plt.plot(theta_path_mgd[:, 0], theta_path_mgd[:, 1], 'g-+', label='MGD')
plt.legend()
plt.show()  # 全局图像
plt.plot(theta_path_bgd[::10, 0], theta_path_bgd[::10, 1], 'b-o', label='BGD')  # ::n 表示以步长为 n 进行切片
plt.plot(theta_path_sgd[::100, 0], theta_path_sgd[::100, 1], 'r-s', label='SGD')
plt.plot(theta_path_mgd[::80, 0], theta_path_mgd[::80, 1], 'g-+', label='MGD')
plt.axis([3.2, 4.5, 2.3, 4.5])
plt.legend()
plt.show()  # 部分数据点的局部图像
# 理论上 BGD 会是笔直到达。SGD 是大范围的浮动，很容易出现偏离。MGD 与 BGD 隔的比较近，在 BGD 的周围浮动。
```

输出

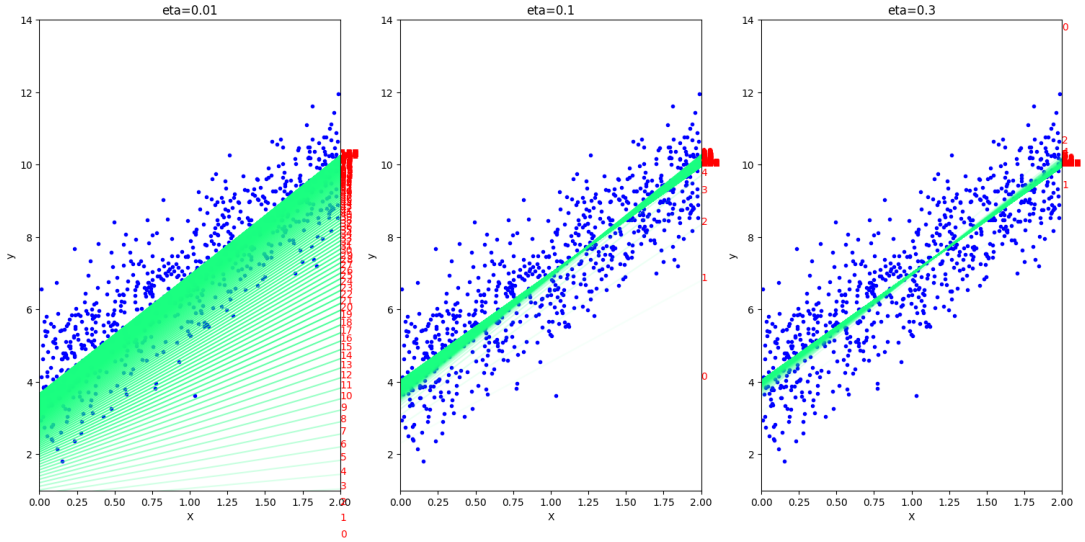
```
公式计算的 θ:  [[4.02650858]
 [2.96474178]]
偏置参数:  [4.02650858]
权重参数:  [[2.96474178]]
MSE 批量梯度下降迭代 10000 次的 θ:  [[4.02650858]
 [2.96474178]]
BGD: [[3.66635721]
 [3.2697089 ]]
SGD: [[3.93608917]
 [3.03300464]]
MGD: [[4.09271048]
 [2.90851744]]
```

图片输出如下：

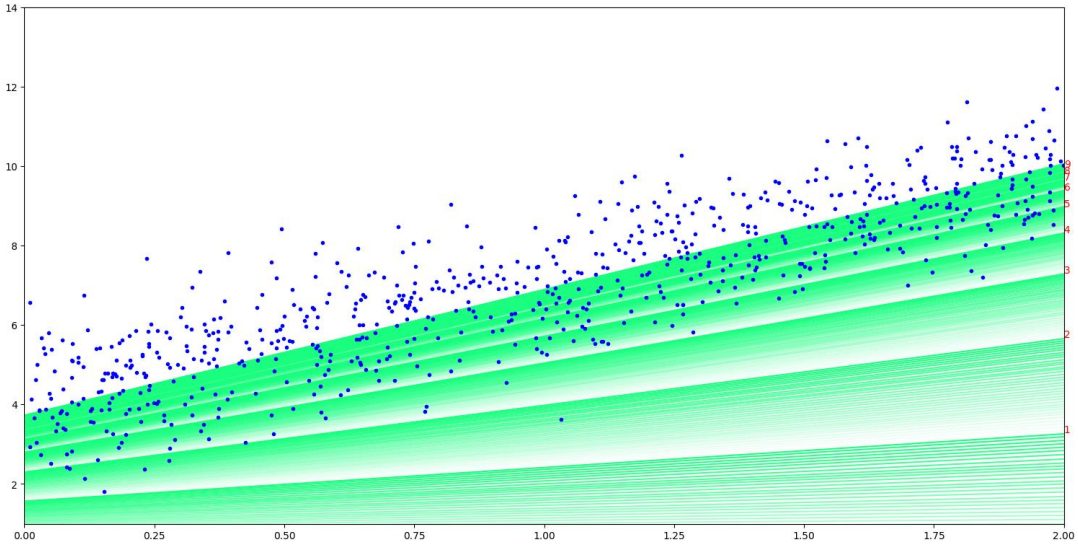
原始数据 & LR 拟合



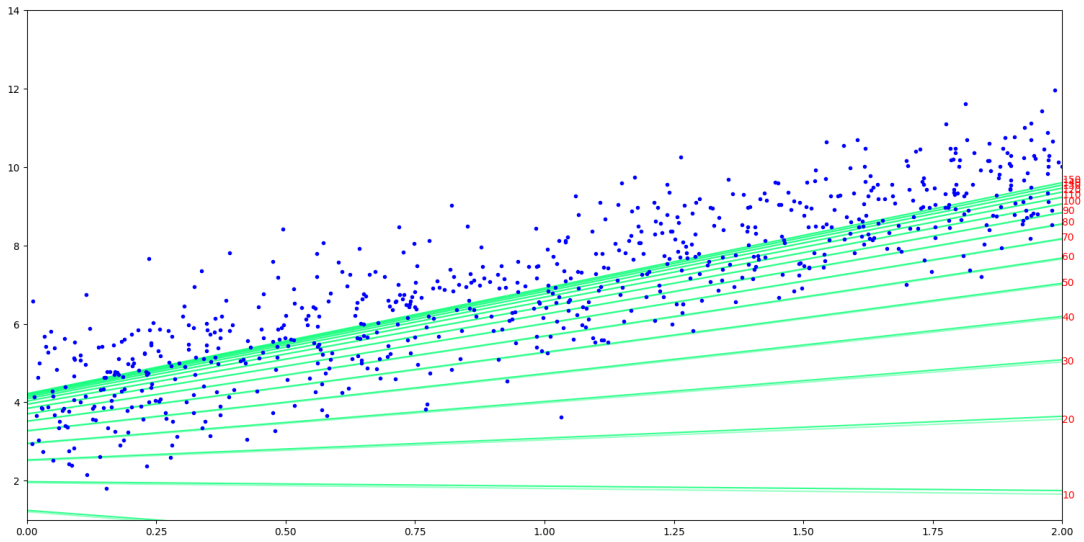
MSE-eta 不同导致的结果不同：



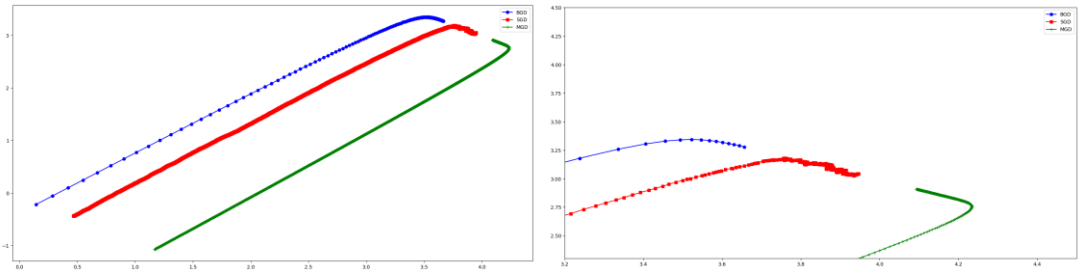
随机梯度下降：



批量梯度下降：



三者对比的完整图 & 部分图：



多项式回归

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
""" ---多项式回归--- """
""" 生成数据集 & 绘图 """
X = 6 * np.random.rand(727, 1) - 3  # 727 个[0, 1)之间均匀分布的随机数 * 6 - 3
y = 0.5*X**2 + X + np.random.randn(727, 1)  # 0.5x^2 + x + 标准正态分布误差项
plt.plot(X, y, 'b.')
plt.axis([-3.2, 3.2, -4, 11])
plt.show()

""" 将多项式化为 LinearRegression """
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)  # 将 X 变为[[X1,X1^2], [X2,X2^2], ... , [Xn,Xn^2]]，相当于将 X^2 变成了一个新的特征
print("X[0]:", X[0], "X_poly[0]", X_poly[0])  # X_poly[i] = (X[i])^2
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
print("偏置参数: ", lin_reg.intercept_)  # 偏置参数 [0.01279778]。即常数项
print("权重参数: ", lin_reg.coef_)  # 权重参数 [[0.98806406 0.4931436 ]]。因为 X_poly 相当于 X 和 X^2，因此系数分别对应一次项和二次项
# 得到拟合曲线为 0.4931436x^2 + 0.98806406x + 0.01279778
X_curve = np.linspace(-3, 3, 100).reshape(100, 1)
X_curve_poly = poly_features.transform(X_curve)  # 从-3 到 3 的 100 个点经过变换(变为 x 与 x^2)后的值
y_curve = lin_reg.predict(X_curve_poly)
plt.plot(X, y, 'b.')
plt.plot(X_curve, y_curve, 'r-')
plt.axis([-3.2, 3.2, -4, 11])
plt.show()

""" degree 不同对结果的影响 """
for style, width, degree in (('g-', 3, 50), ('r-', 2, 2), ('y--', 2, 1)):
    poly_features = PolynomialFeatures(degree=degree, include_bias=False)
    std = StandardScaler()
    lin_reg = LinearRegression()
    # 流水线车间 Pipeline 包含三个流水线：poly_features,StandardScaler,LinearRegression
```

```
    polynomial_reg = Pipeline([('poly_features', poly_features),  # 多项式特征生成
                               ('StandardScaler', std),  # 标准化
                               ('LinearRegression', lin_reg)])  # 线性回归
    polynomial_reg.fit(X, y)
    y_curve_ComparativeExperiments = polynomial_reg.predict(X_curve)
    plt.plot(X_curve, y_curve_ComparativeExperiments, style, label='degree:'+str(degree), linewidth=width)
plt.plot(X, y, 'b.')
plt.axis([-3.2, 3.2, -4, 11])
plt.legend()
plt.show()
```

Logistic 回归

```
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

Iris = load_iris()
X = Iris.data
Y = Iris.target
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,
random_state=0)
LR = LogisticRegression(penalty='l2', solver='newton-cg',
multi_class='multinomial')
LR.fit(x_train,y_train)
print("Logistic Regression 模型训练集的准确率：%.3f" % LR.score(x_train, y_train))
print("Logistic Regression 模型测试集的准确率：%.3f" % LR.score(x_test, y_test))
y_hat = LR.predict(x_test)
print(classification_report(y_test, y_hat, target_names=Iris.target_names))
```

输出

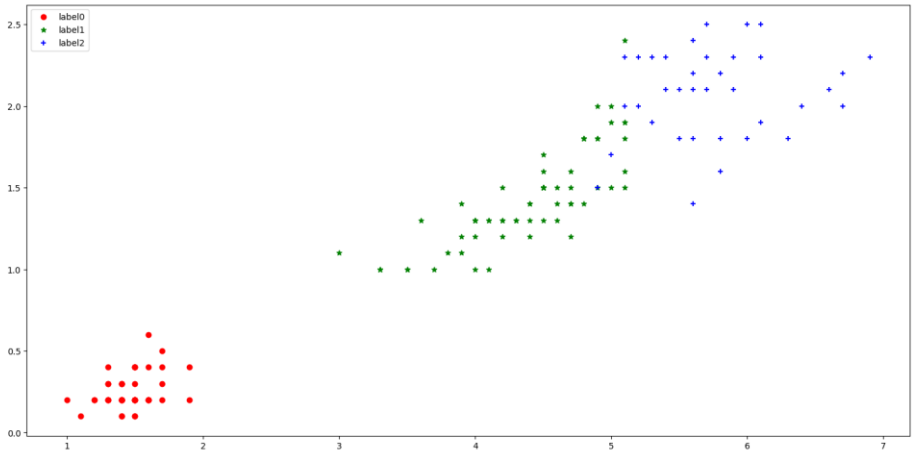
```
Logistic Regression 模型训练集的准确率：0.981
Logistic Regression 模型测试集的准确率：0.978
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        16
  versicolor       1.00      0.94      0.97        18
   virginica       0.92      1.00      0.96        11

    accuracy                           0.98        45
   macro avg       0.97      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45
```

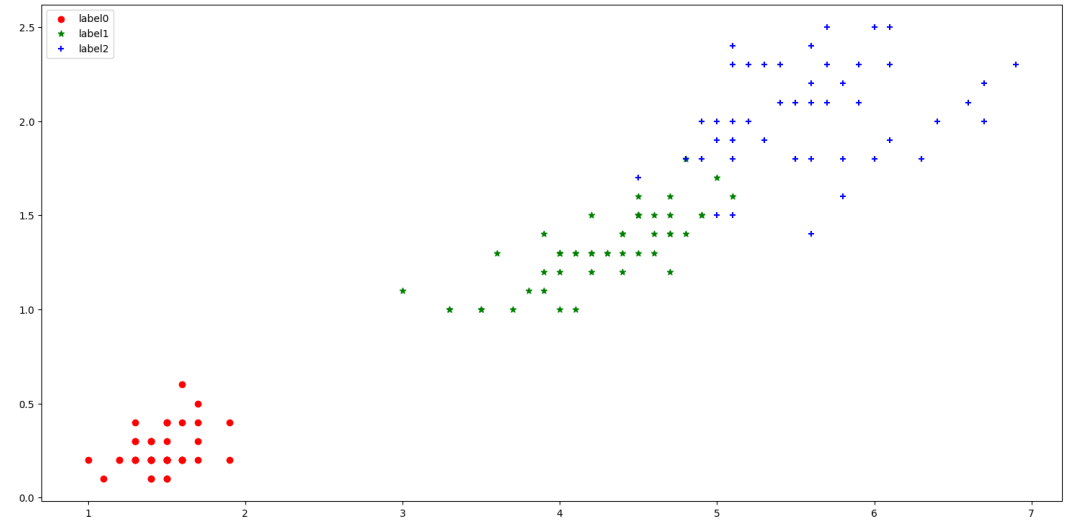
8.聚类

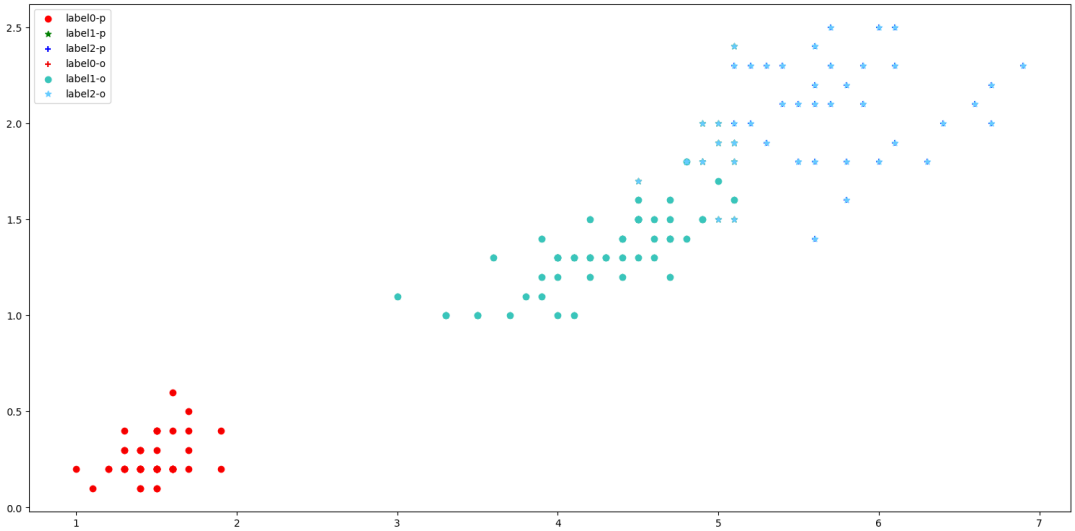
K-means

```
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

Iris = load_iris()
X = Iris.data
Y = Iris.target

estimator = KMeans(n_clusters=3, n_init=10)  # n_init 是 KMeans 算法中用于指定随机初始化的次数的参数。
estimator.fit(X)
label_predict = estimator.labels_  # 是一个 list，包含对应的聚类标签(0,1,2)

# 绘制预测数据
xp0 = X[label_predict == 0]  # xp0 代指 predict 的 label 是 0 的 x
xp1 = X[label_predict == 1]
xp2 = X[label_predict == 2]
plt.scatter(xp0[:, 2], xp0[:, 3], c='red', marker='o', label='label0')
plt.scatter(xp1[:, 2], xp1[:, 3], c='green', marker='*', label='label1')
plt.scatter(xp2[:, 2], xp2[:, 3], c='blue', marker='+', label='label2')
plt.legend(loc=2)  # 添加图例(loc=2 代表左上角绘制)
plt.show()

# 绘制原始数据
xo0 = X[Y == 0]  # xo0 代指 original label 是 0 的 x
xo1 = X[Y == 1]
xo2 = X[Y == 2]
plt.scatter(xo0[:, 2], xo0[:, 3], c='red', marker='o', label='label0')
plt.scatter(xo1[:, 2], xo1[:, 3], c='green', marker='*', label='label1')
plt.scatter(xo2[:, 2], xo2[:, 3], c='blue', marker='+', label='label2')
plt.legend(loc=2)
plt.show()

# 比较绘制
plt.scatter(xp0[:, 2], xp0[:, 3], c='red', marker='o', label='label0-p')
plt.scatter(xp1[:, 2], xp1[:, 3], c='green', marker='*', label='label1-p')
plt.scatter(xp2[:, 2], xp2[:, 3], c='blue', marker='+', label='label2-p')
plt.scatter(xo0[:, 2], xo0[:, 3], c='#EE0000', marker='+', label='label0-o')
plt.scatter(xo1[:, 2], xo1[:, 3], c='#39C5BB', marker='o', label='label1-o')
plt.scatter(xo2[:, 2], xo2[:, 3], c='#66CCFF', marker='*', label='label2-o')
plt.legend(loc=2)
plt.show()
```

输出（依次为预测、原始、预测和原始叠加）







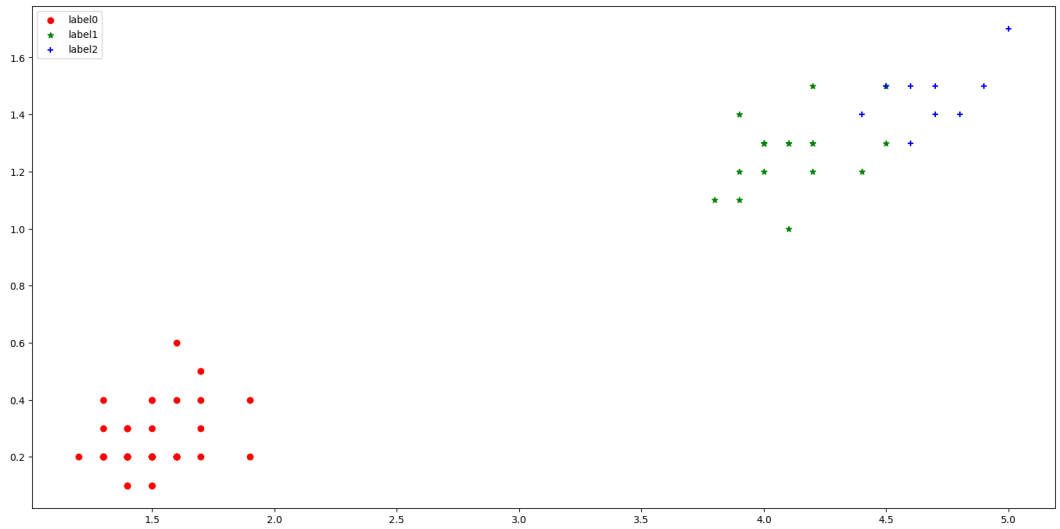
当聚类数量较多不便书写时可考虑以下两种方法:（以绘制原始数据的 target 值的图为例）

```
for target_value in set(Y):
    x = X[Y == target_value]
    if target_value == 0:
        plt.scatter(x[:, 2], x[:, 3], c='red', marker='o', label=f'Target
{target_value}')
    elif target_value == 1:
        plt.scatter(x[:, 2], x[:, 3], c='green', marker='*', label=f'Target
{target_value}')
    else:
        plt.scatter(x[:, 2], x[:, 3], c='blue', marker='+', label=f'Target
{target_value}')

# 定义颜色和标记对应关系
colors = ['red', 'green', 'blue']
markers = ['o', '*', '+']
for target_value in set(Y):
    x = X[Y == target_value]
    color = colors[target_value]
    marker = markers[target_value]
    label = f'Target {target_value}'
    plt.scatter(x[:, 2], x[:, 3], c=color, marker=marker, label=label)
```

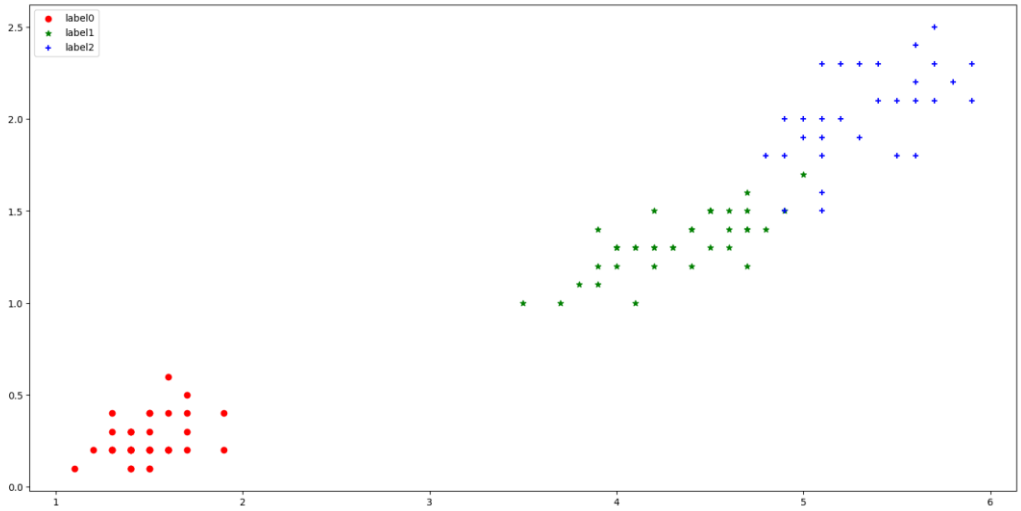
DBSCAN

```
from sklearn.datasets import load_iris
from sklearn.cluster import DBSCAN
import matplotlib.pyplot as plt

Iris = load_iris()
X = Iris.data
Y = Iris.target

dbscan = DBSCAN(eps=0.4, min_samples=9)  # 半径 0.4 内最少要有 9 个样本
dbscan.fit(X)
label_predict = dbscan.labels_  # 是一个 list，包含对应的聚类标签(0,1,2)

# 绘制预测数据
xp0 = X[label_predict == 0]  # xp0 代指 predict 的 label 是 0 的 x
xp1 = X[label_predict == 1]
xp2 = X[label_predict == 2]
plt.scatter(xp0[:, 2], xp0[:, 3], c='red', marker='o', label='label0')
plt.scatter(xp1[:, 2], xp1[:, 3], c='green', marker='*', label='label1')
plt.scatter(xp2[:, 2], xp2[:, 3], c='blue', marker='+', label='label2')
plt.legend(loc=2)  # 添加图例(loc=2 代表左上角绘制)
plt.show()
```

输出



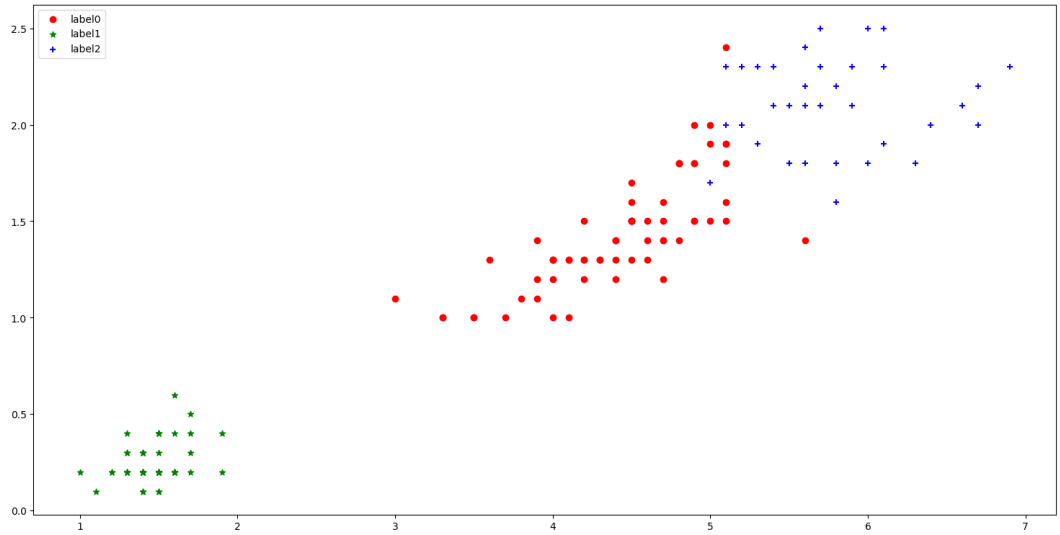
会发现和想要的结果不太一样。更改参数为 eps=0.4, min_samples=4，输出：



AGENS

```
from sklearn.datasets import load_iris
from sklearn.cluster import AgglomerativeClustering
import matplotlib.pyplot as plt

Iris = load_iris()
X = Iris.data
Y = Iris.target

AGENS = AgglomerativeClustering(linkage='ward', n_clusters=3)  # 以 ward 方差最小化的方式来计算簇的距离，分为 3 个簇。
AGENS.fit(X)
label_predict = AGENS.labels_  # 是一个 list，包含对应的聚类标签(0,1,2)

# 绘制预测数据
xp0 = X[label_predict == 0]  # xp0 代指 predict 的 label 是 0 的 x
xp1 = X[label_predict == 1]
xp2 = X[label_predict == 2]
plt.scatter(xp0[:, 2], xp0[:, 3], c='red', marker='o', label='label0')
plt.scatter(xp1[:, 2], xp1[:, 3], c='green', marker='*', label='label1')
plt.scatter(xp2[:, 2], xp2[:, 3], c='blue', marker='+', label='label2')
plt.legend(loc=2)  # 添加图例(loc=2 代表左上角绘制)
plt.show()
```

输出



集成算法

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris
from sklearn.ensemble import AdaBoostClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier

Iris = load_iris()
X = Iris.data
Y = Iris.target
clf = AdaBoostClassifier(n_estimators=100)  # 迭代次数为 100
scores = cross_val_score(clf, X, Y)  # 交叉验证
print('AdaBoost 准确率：', scores.mean())

bagging = BaggingClassifier(KNeighborsClassifier(), max_samples=0.5,
max_features=0.5)  # 结合 KNN 分类器，每个基分类器抽样样本数量和特征数量的最大比例为 0.5
scores = cross_val_score(bagging, X, Y)
print('bagging 准确率：', scores.mean())

clf = RandomForestClassifier(n_estimators=10, max_features=2)  # 决策树的数量为 10
棵，每棵树最大的特征数量为 2
scores = cross_val_score(clf, X, Y)
print('RF 准确率：', scores.mean())
```

输出

```
AdaBoost 准确率：  0.9466666666666665
bagging 准确率：  0.9533333333333334
RF 准确率：  0.9533333333333334
```

性能比较

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import VotingClassifier

categories = ['alt.atheism', 'soc.religion.christian', 'comp.graphics', 'sci.med']
twenty_train = fetch_20newsgroups(subset='train', categories=categories,
shuffle=True, random_state=42)
count_vect = CountVectorizer()
X_train_counts = count_vect.fit_transform(twenty_train.data)
tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
twenty_test = fetch_20newsgroups(subset='test', categories=categories,
shuffle=True, random_state=42)
X_test_counts = count_vect.transform(twenty_test.data)
X_test_tfidf = tfidf_transformer.transform(X_test_counts)


def Vot_clf(voting_type):
    log_clf = LogisticRegression(penalty='l2', solver='newton-cg')
    tree_clf = DecisionTreeClassifier(criterion='entropy')
    svm_clf = SVC(kernel='linear', probability=True)
    return VotingClassifier([('lr', log_clf), ('tree', tree_clf), ('svc', svm_clf)],
voting=voting_type, weights=[2, 1, 3])


def Acc_of_clf(voting_type):
    log_clf = LogisticRegression()
```

```
    tree_clf = DecisionTreeClassifier(criterion='entropy')
    svm_clf = SVC(kernel='linear', probability=True)
    vot_clf = Vot_clf(voting_type)
    for clf in (log_clf, tree_clf, svm_clf, vot_clf):
        clf.fit(X_train_tfidf, twenty_train.target)
        print(clf.__class__.__name__, "准确率%.4f" % (clf.score(X_test_tfidf,
twenty_test.target)))


Acc_of_clf('soft')
```

输出

```
LogisticRegression 准确率 0.8975
DecisionTreeClassifier 准确率 0.6897
SVC 准确率 0.9208
VotingClassifier 准确率 0.9288
```

用于比较逻辑回归、决策树、支持向量机和集成投票分类器在 20news 文本数据集上的性能。

神经网络

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

Iris = load_iris()
X = Iris.data
Y = Iris.target
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.3)
model = MLPClassifier(max_iter=1000)
model.fit(x_train, y_train)
print("神经网络模型训练集的准确率：%.3f" % model.score(x_train, y_train))
print("神经网络模型测试集的准确率：%.3f" % model.score(x_test, y_test))
```

输出

```
神经网络模型训练集的准确率：0.971
神经网络模型测试集的准确率：0.978
```