

Python与数据科学导论-08

—— Pandas基础



2022/03/28 胡俊峰

北京大学计算机学院



内容

- 聚类及评价（续）
- 非负矩阵分解
- Pandas introduction

聚类的质量评价

- 指标：纯度 (Purity) 和F值 (F-measure)
- 标准答案：一般是人工分好类的文档集合

纯度

- 对于聚类后形成的任意类别 r ，聚类的纯度定义为

$$P(S_r) = \frac{1}{n_r} \max(n_r^i)$$

- 整个聚类结果的纯度定义为

$$\text{Purity} = \sum_{r=1}^k \frac{n_r}{n} P(S_r)$$

- n_r^i : 属于预定义类 i 且被分配到第 r 个聚类的文档个数
- n_r : 第 r 个聚类类别中的文档个数

F值

- F值：准确率（precision）和召回率（recall）的调和平均数
- $\text{precision}(i, r) = n_r^i / n_r$
- $\text{recall}(i, r) = n_r^i / n_i$
- n_r^i ：属于预定义类i且被分配到第r个聚类的文档个数
- n_r ：第r个聚类类别中的文档个数
- n_i ：预定义类别i中的文档个数

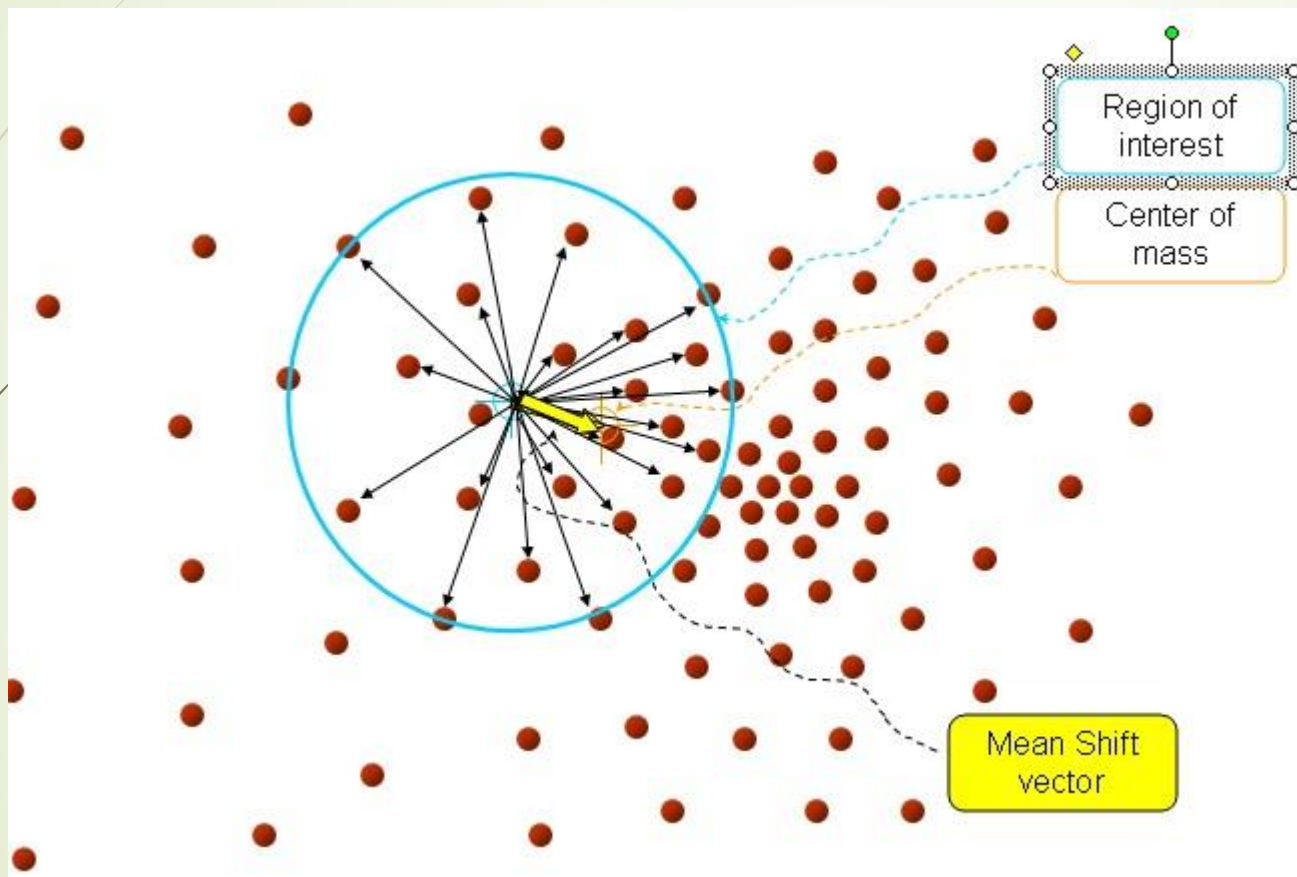
F值

- 聚类 r 和类别 i 之间的 f 值计算如下:
- $$f(i, r) = \frac{2 \times recall(i, r) \times precision(i, r)}{precision(i, r) + recall(i, r)}$$
- 最终聚类结果的评价函数为
- $$F = \sum_i \frac{n_i}{n} \max\{f(i, r)\}, \quad n \text{ 是所有文档的个数}$$

K-means 的改进

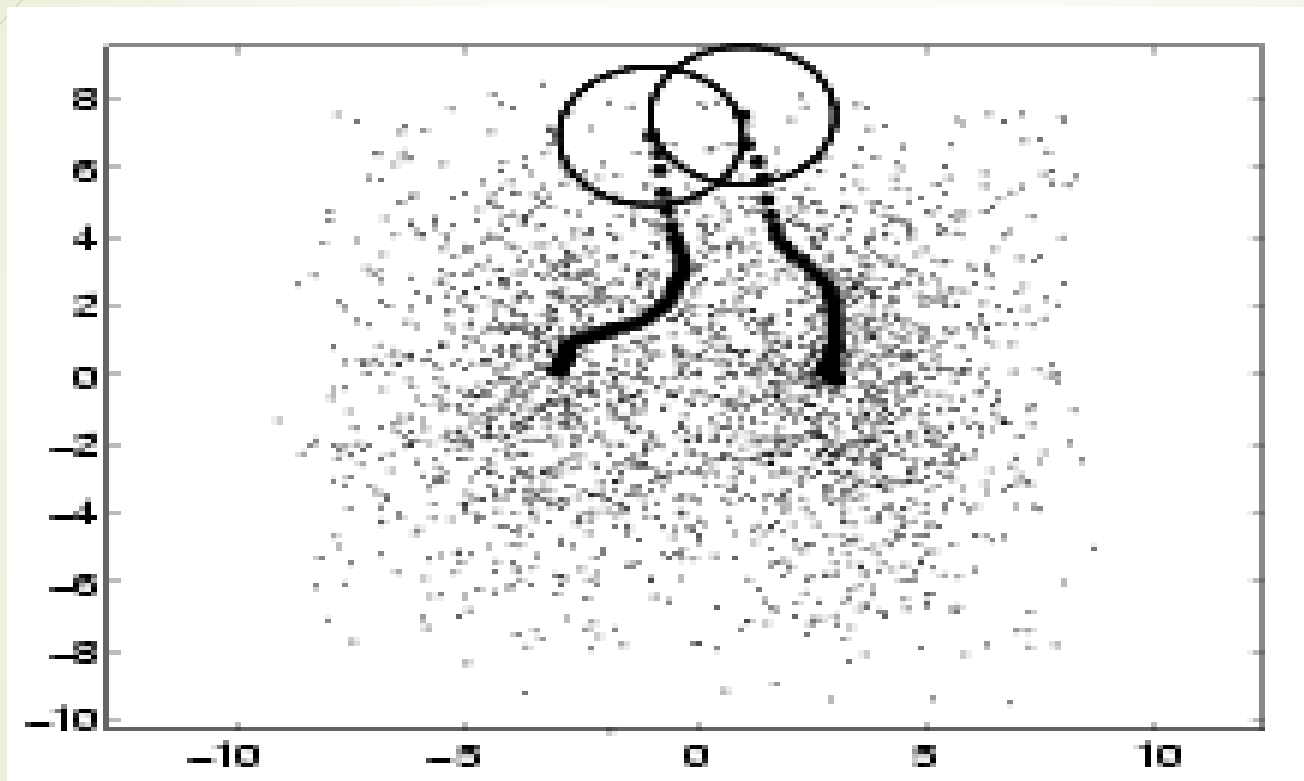
- 确定K
 - 对于不同的K都进行多次尝试聚类，取效果最好的
- 合理选定初始种子
- Bi-secting k-means

Mean-shift算法



在核函数约束下呈现的按梯度贪心的收敛轨迹

mean shift trajectories



Window tracks signify the steepest ascent directions

距离加权方案与核函数 (kernel)

$$P(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n K(\mathbf{x} - \mathbf{x}_i)$$

A function of some finite number of data points
 $\mathbf{x}_1 \dots \mathbf{x}_n$

Examples:

- Epanechnikov Kernel

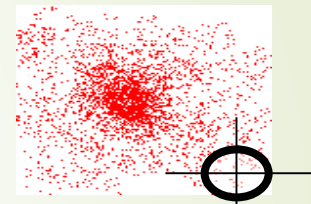
$$K_E(\mathbf{x}) = \begin{cases} c(1 - \|\mathbf{x}\|^2) & \|\mathbf{x}\| \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

- Uniform Kernel

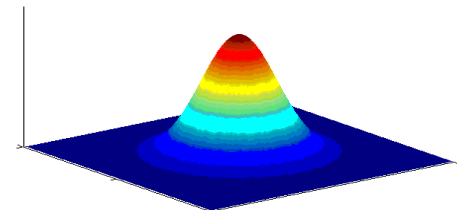
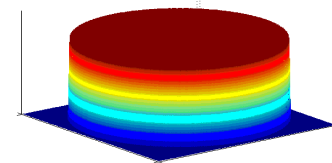
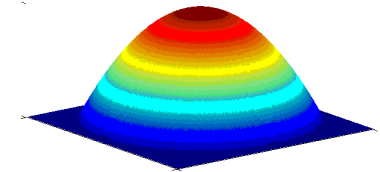
$$K_U(\mathbf{x}) = \begin{cases} c & \|\mathbf{x}\| \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

- Normal Kernel

$$K_N(\mathbf{x}) = c \cdot \exp\left(-\frac{1}{2}\|\mathbf{x}\|^2\right)$$

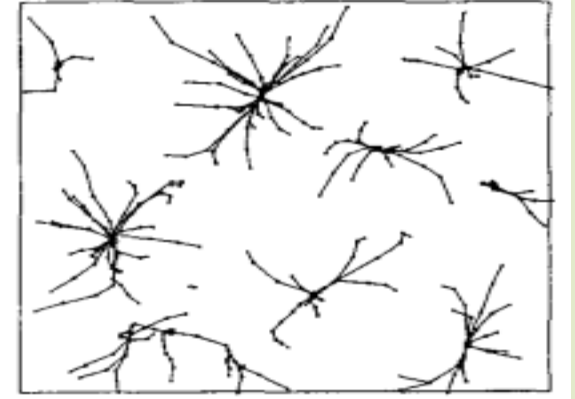
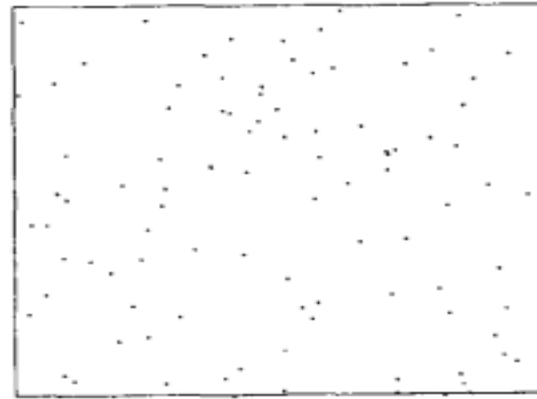


Data

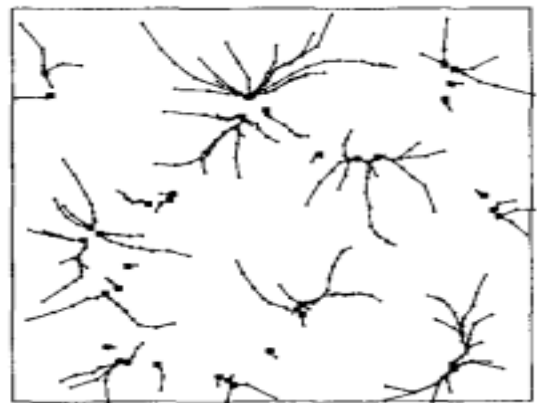


Mean shift trajectories with different kernel

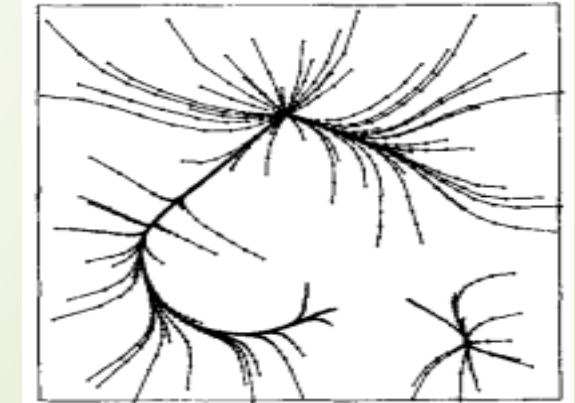
- Same sample space
 - ① Uniform Kernel
 - ② truncated Normal Kernel
 - ③ nontruncated Normal Kernel
- 可以用来实现无先验K值的k-means



(1)



(2)



(3)

矩阵分解：

`sklearn.decomposition`: Matrix Decomposition

The `sklearn.decomposition` module includes matrix decomposition algorithms, including among others PCA, NMF or ICA. Most of the algorithms of this module can be regarded as dimensionality reduction techniques.

User guide: See the [Decomposing signals in components \(matrix factorization problems\)](#) section for further details.

<code>decomposition.DictionaryLearning([...])</code>	Dictionary learning.
<code>decomposition.FactorAnalysis([n_components, ...])</code>	Factor Analysis (FA).
<code>decomposition.FastICA([n_components, ...])</code>	FastICA: a fast algorithm for Independent Component Analysis.
<code>decomposition.IncrementalPCA([n_components, ...])</code>	Incremental principal components analysis (IPCA).
<code>decomposition.KernelPCA([n_components, ...])</code>	Kernel Principal component analysis (KPCA).
<code>decomposition.LatentDirichletAllocation([...])</code>	Latent Dirichlet Allocation with online variational Bayes algorithm.
<code>decomposition.MinibatchDictionaryLearning([...])</code>	Mini-batch dictionary learning.
<code>decomposition.MinibatchSparsePCA([...])</code>	Mini-batch Sparse Principal Components Analysis.
<code>decomposition.NMF([n_components, init, ...])</code>	Non-Negative Matrix Factorization (NMF).
<code>decomposition.PCA([n_components, copy, ...])</code>	Principal component analysis (PCA).
<code>decomposition.SparsePCA([n_components, ...])</code>	Sparse Principal Components Analysis (SparsePCA).
<code>decomposition.SparseCoder(dictionary, *[, ...])</code>	Sparse coding.
<code>decomposition.TruncatedSVD([n_components, ...])</code>	Dimensionality reduction using truncated SVD (aka LSA).

```
n_components = 30 # Extracting the PCA n_components
pca = PCA(n_components=n_components).fit(X_s) # 拟合最小损失的30维PCA空间, SKlearn版, 中心化后做SVD
eigenvectors = pca.components_.reshape(n_components, 28, 28) # 把特征向量转成二维图片
```

可解释方差占比

```
variance_ratio = pca.explained_variance_ratio_
variance_ratio[:5]
```

```
array([0.09634413, 0.07171227, 0.0604996 , 0.05463821, 0.0480601 ])
```

```
svd = TruncatedSVD(n_components).fit(X_s) # 降维奇异值分解
```

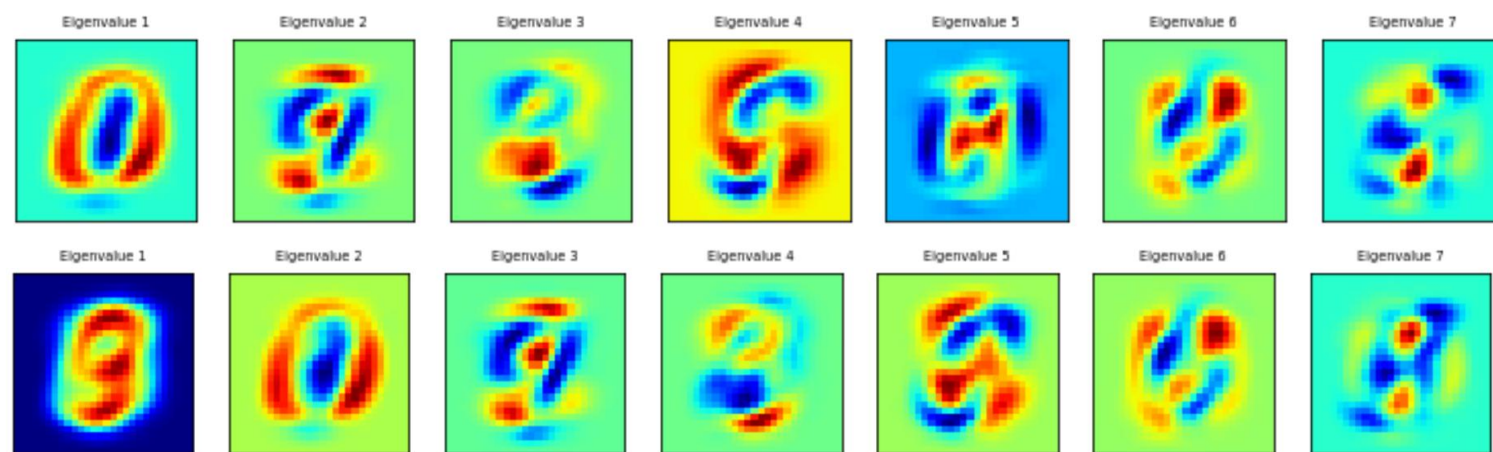
```
singularvectors = svd.components_.reshape(n_components, 28, 28) # 把奇异值向量转成二维图片
```

Extracting the SVD variance_ratio of components

```
variance_ratio = svd.explained_variance_ratio_
variance_ratio[:5]
```

```
array([0.05710067, 0.08192285, 0.07156619, 0.06023309, 0.05367006])
```

PCA vs SVD



样本聚类 与非负矩阵分解 $X \approx F * G^T$

分解后的F矩阵中每一列对应一个特征的向量，而 G^T 矩阵的每一列对F中的特征向量组进行线性组合来编码X中的一个样本

如果 G^T 中的每一列且仅有一个非0元，则X中每个元素被F中每一列近似表出，此时F的每个列向量可以认为是一个类质心。同时自然能得出 G^T 的每一行之间是正交的。 G^T 同一行中所有非零元对应的 X_i 被在变换后被投影到同一个点（同一类）（VQ）

NMF非负矩阵分解

- 目标：将 $n * m$ 的矩阵 V 分解为两个非负矩阵的乘积，即： $V \approx W * H$
- $W: n * r, H: r * m$
- r 越小，代表压缩比会更高，也会意味着更大的重建损失

$$J(W, H) = \frac{1}{2} \sum_{i,j} [V_{ij} - (WH)_{ij}]^2$$

- 非负矩阵分解可以采用梯度下降法求解，但结果不唯一。想得到好的特征分解需要加入先验或进行多次尝试。

K-means聚类 与 非负矩阵 (NMF) 分解

➡ $A_i' = u_i * v$ ($u_{ij} == 1$) ; ($u_{ik} (k \neq j) == 0$) (QV分解)

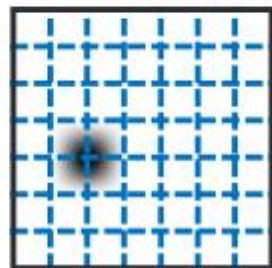
➡ Loss (A-A')

初始化：随机选取样本做质心
聚类结果为特定类型的公共脸

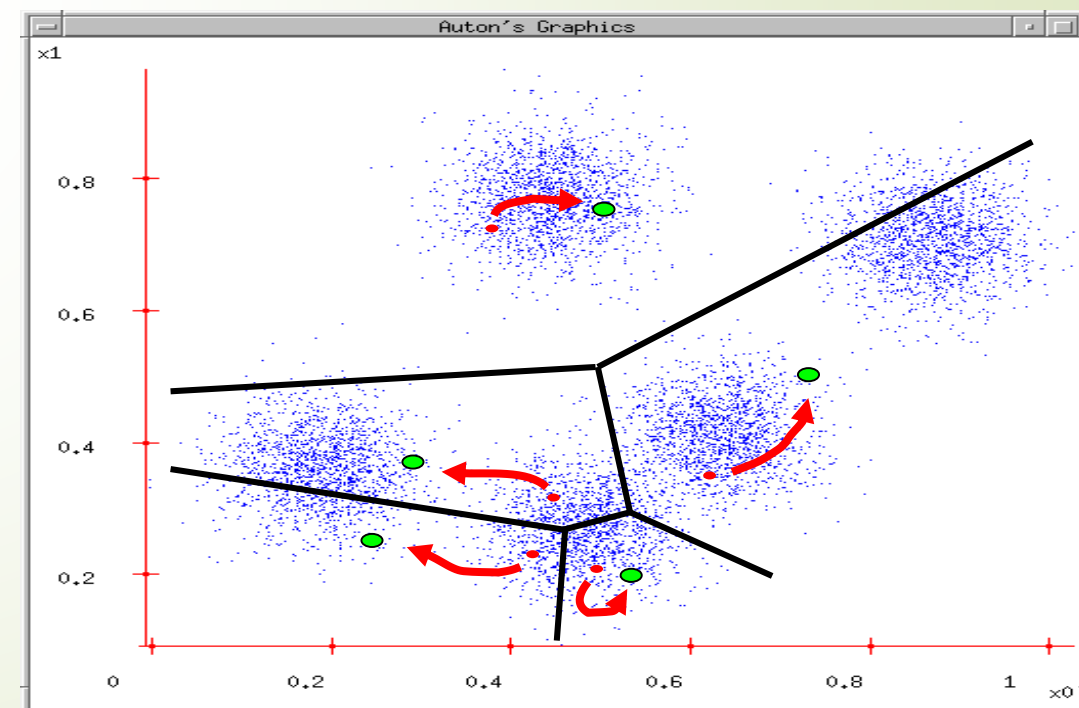
VQ



×



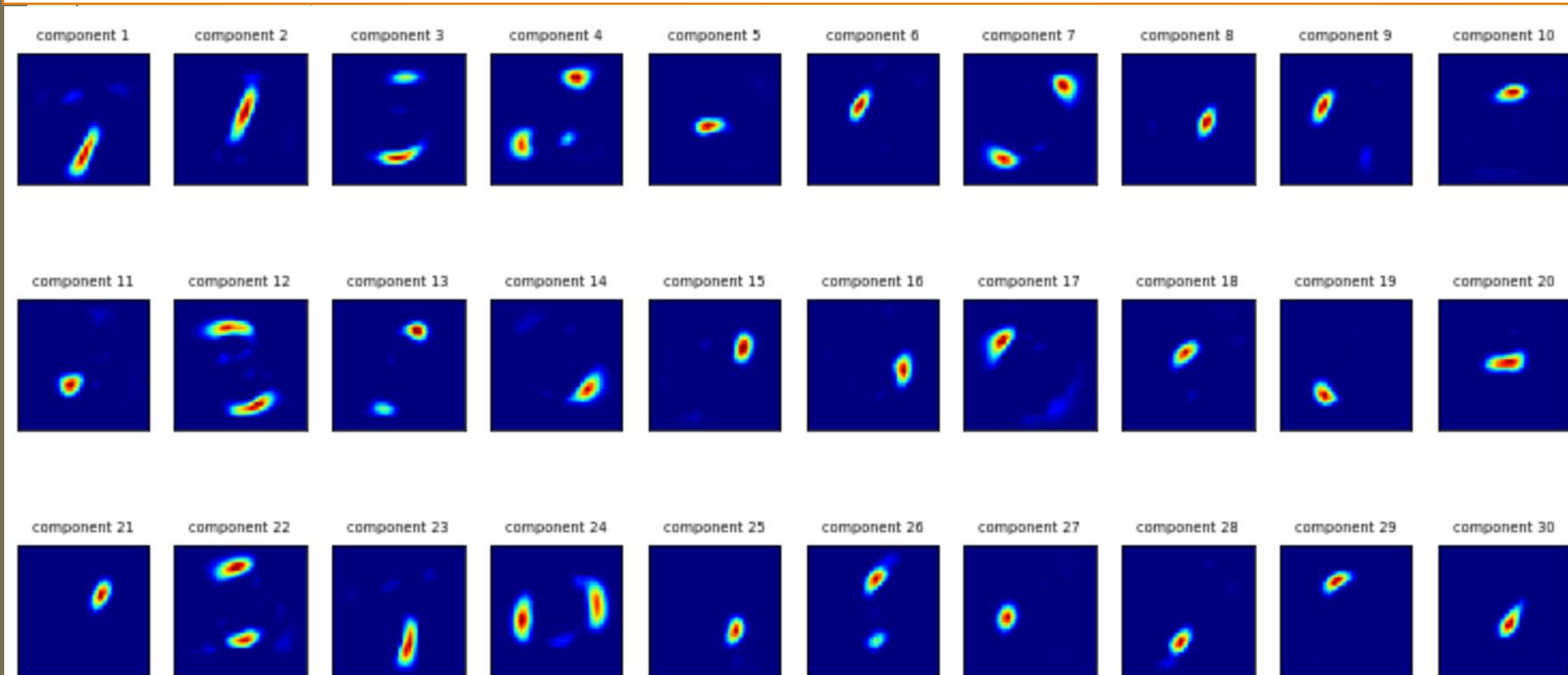
=



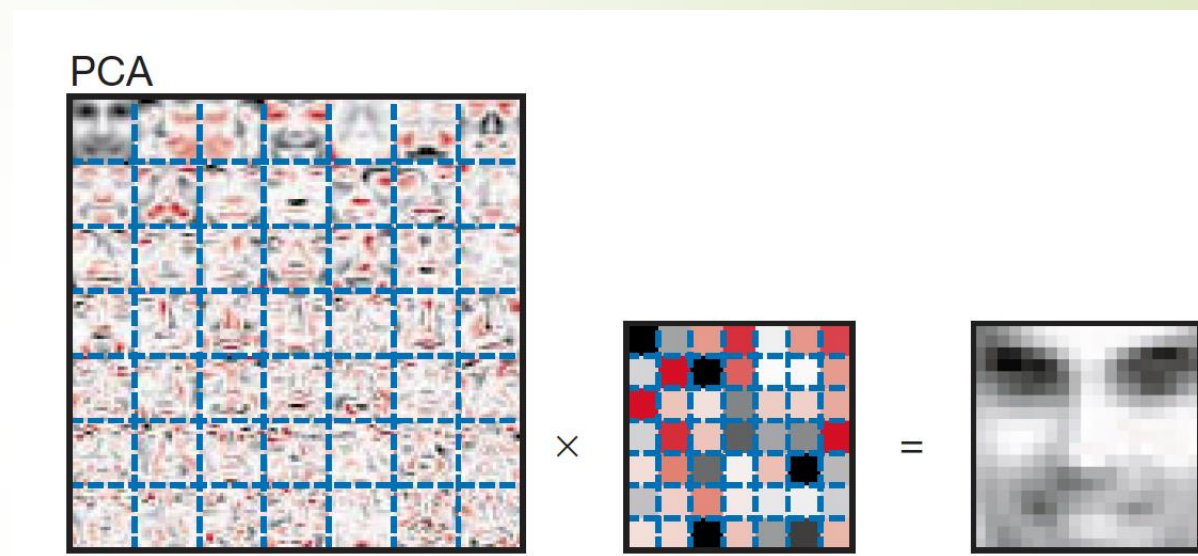
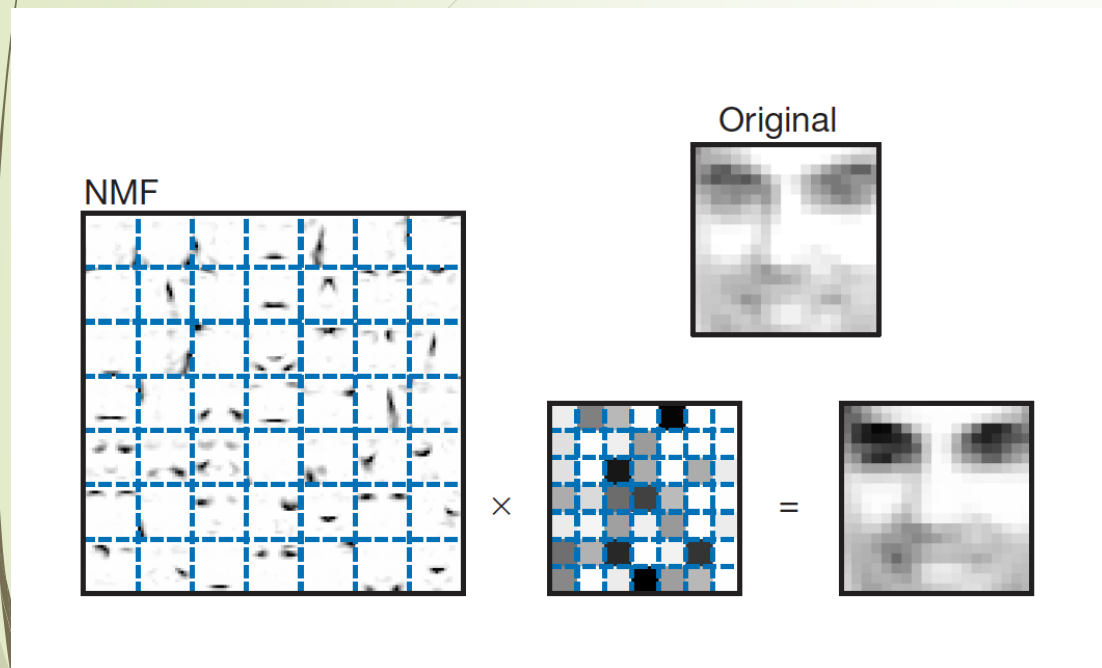
NMF特征分解:

```
from sklearn.decomposition import NMF
nmf = NMF(30, init = 'nndsvda').fit(X_s)
nmfvectors = nmf.components_.reshape(n_components, 28, 28) # 把nmf 'dictionary' 转成二维图片
reconstruction_err = nmf.reconstruction_err_
print(reconstruction_err)
```

86601.88621066371



NMF稀疏特征分解 vs PCA压缩编码：

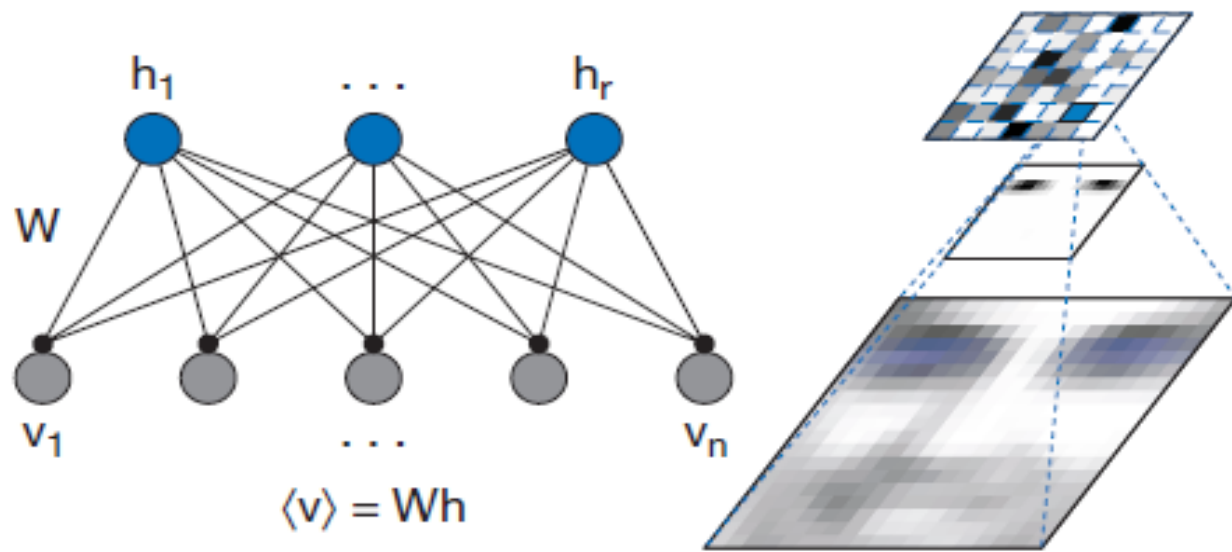


标红的为负数

可以看出PCA学到特征更抽象，NMF学到的更能体现部分特征。
NMF无论是feature还是encoding，矩阵都比较稀疏。

非负矩阵分解的深层意义：

- ▶ 矩阵分解的基矩阵和Encoding矩阵都是非负的
- ▶ 矩阵分解得到的基矩阵和Encoding矩阵是稀疏的
- ▶ Encoding矩阵中的非零元可以看做对于部分**隐藏特征**的“激活”
- ▶ 考虑到对隐藏特征的激活权重可以看成是隐藏特征在样本上的生成权重。则该模型就可以理解为样本是有多个隐含的概率分布生成出来的。



Pandas概述

- 是以记录为数据单元，记录的序列（表）为操作单元的数据处理平台
- 支持索引，多键值索引下的表间运算（合并）
- 支持按行的向量广播计算、按属性的聚合运算
- 索引值唯一，属性值可重复，属性值可实现表间连接计算

Pandas — Panel data analysis

- 序列: indexed list
- 多通道序列: record list
- 多字段二维表
- 表关联运算
- Pandas的统计功能与应用

The Pandas Series Object —— 序列

A Pandas Series is a one-dimensional array of indexed data. It can be created from a list or array as

缺省情况类似excel的表格，自动维护标号索引

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])  
print(data)
```

```
data.index
```

```
0    0.25
```

```
1    0.50
```

```
2    0.75
```

```
3    1.00
```

```
dtype: float64
```

```
RangeIndex(start=0, stop=4, step=1)
```


与数组类似，支持下标切片访问操作

```
1 data.values
```

```
array([ 0.25,  0.5 ,  0.75,  1.  ])
```

The index is an array-like object of type `pd. Index`

```
1 data[1]
```

```
0.5
```

```
1 data[1:3] ← 下标切片访问
```

```
1    0.50
```

```
2    0.75
```

```
dtype: float64
```

也可以指定可哈希的索引项，类似dict

```
1 data = pd.Series([0.5, 0.25, 1.75, 1.0],  
2                  index=['a', 'b', 'c', 'd']) ←  
3 print(data)  
4 print(data.sort_values())  
5 data['b'] ←
```

```
a    0.50  
b    0.25  
c    1.75  
d    1.00  
dtype: float64  
b    0.25  
a    0.50  
d    1.00  
c    1.75  
dtype: float64
```

0.25

非连续的索引项也没可以，但一般建议避免非连续数字

```
1 data = pd.Series([0.25, 0.5, 0.75, 1.0],  
2                   index=[2, 5, 3, 7])  
3 data[5]
```

0.5

print(data[1:3])

5	0.50
3	0.75

```
1 data[data>0.7] * 2
```

类似numpy，可以通过布尔条件生成下标访问序列

3 1.5

7 2.0

dtype: float64

Indexers: loc, iloc, and ix

按位置索引

These slicing and indexing conventions can be a source of confusion. For example, if your `Series` has an explicit integer index, an indexing operation such as `data[1]` will use the explicit indices, while a slicing operation like `data[1:3]` will use the implicit Python-style index.

```
1 data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
2 data
```

```
1    a
3    b
5    c
dtype: object
```

```
1 # explicit index when indexing
2 data[1]
```

'a'

```
1 # implicit index when slicing
2 data[1:3]
```

```
3    b
5    c
dtype: object
```

```
1 print(data.loc[1])
2 print(data.iloc[1])
3
```

a
b

```
1 print(0.75 in data)
2 0.75 in data.values
```

False

True

```
1 for i in data.values:
2     print(i)
```


← 迭代器，也要指明具体对象

0.25
0.5
0.75
1.0


True

```
for k in data.index:
    print(data[k])
```

0.5
0.25
1.75
1.0



```
1 a = pd.Series([2, 4, 6])
2 b = pd.Series({2:'a', 1:'b', 3:'c'})
3 print(b[1])
4 2 in b
```



词典数据初始化序列

b

True

```
1 for i in b:
2     print (i)
```



直接in不行，迭代器默认value

a

b

c

```
1 sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
2 obj3 = pd.Series(sdata)
3 print(obj3)
4 states = ['California', 'Ohio', 'Oregon', 'Texas']
5 obj4 = pd.Series(sdata, index=states) ← 可以为一个序列指定新索引，类似索引重建
6 obj4
```

```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

```
California    NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

由于"California"所对应的sdata值找不到，所以其结果就为NaN（即“非数字”（not a number），在pandas中，它用于表示缺失或NA值）。因为‘Utah’不在states中，它被从结果中除去。


```
1 # Series最重要的一个功能是，它会根据运算的索引标签自动对齐数据
2 # 关于数据对齐功能如果你使用过数据库，可以认为是类似join的操作
3 obj3+obj4
```

```
California      NaN
Ohio            70000.0
Oregon          32000.0
Texas           142000.0
Utah            NaN
dtype: float64
```

```
1 obj3 - obj4
```

```
California      NaN
Ohio            0.0
Oregon          0.0
Texas           0.0
Utah            NaN
dtype: float64
```

对应元素element wise运算，非对应元素NaN

The Pandas DataFrame Object

- 视角1：多个对齐的序列（series）的组合（record）
- 视角2：多维度的 Numpy array 支持 多维度索引
- 视角3：多帧数据的序列，每帧数据是一个Numpy array

```
1 population_dict = {'California': 38332521,
2                     'Texas': 26448193,
3                     'New York': 19651127,
4                     'Florida': 19552860,
5                     'Illinois': 12882135}
6 population = pd.Series(population_dict)
7
8 area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
9              'Florida': 170312, 'Illinois': 149995}
10 area = pd.Series(area_dict)
```

```
1 states = pd.DataFrame({'population': population, 'area': area})
2 states
```

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

- 1、词典到序列数据
- 2、多序列合并生成dataframe

```

1 population_dict = {'California': 38332521, 'Texas': 26448193,
2                   'New York': 19651127, 'W.DC': 11000000}
3 population = pd.Series(population_dict)
4
5 area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
6             'Florida': 170312, 'Illinois': 149995}
7 area = pd.Series(area_dict)

```

```

1 states = pd.DataFrame({'population': population, 'area': area})
2 states

```

	population	area
California	38332521.0	423967.0
Florida	NaN	170312.0
Illinois	NaN	149995.0
New York	19651127.0	141297.0
Texas	26448193.0	695662.0
W.DC	11000000.0	NaN

索引-数据 与 索引合并（非对齐情况）：
索引扩展，数据用NaN填充

```
1 print(states.index)
2 print(states.columns)
3 for i in states.columns:
4     print(states[i])
```

行列索引（本质上是多重索引）

```
Index(['California', 'Florida', 'Illinois', 'New York', 'Texas', 'W.DC'], dtype='object')
```

```
Index(['population', 'area'], dtype='object')
```

都是索引对象，逻辑上对等，类似矩阵维度

```
California      38332521.0
```

```
Florida          NaN
```

```
Illinois         NaN
```

```
New York        19651127.0
```

```
Texas            26448193.0
```

```
W.DC             11000000.0
```

```
Name: population, dtype: float64
```

```
California      423967.0
```

```
Florida         170312.0
```

```
Illinois        149995.0
```

```
New York        141297.0
```

```
Texas           695662.0
```

```
W.DC            NaN
```

```
Name: area, dtype: float64
```

表5-1：可以输入给DataFrame构造器的数据

类型	说明
二维ndarray	数据矩阵，还可以传入行标和列标
由数组、列表或元组组成的字典	每个序列会变成DataFrame的一列。所有序列的长度必须相同
NumPy的结构化/记录数组	类似于“由数组组成的字典”
由Series组成的字典	每个Series会成为一列。如果没有显式指定索引，则各Series的索引会被合并成结果的行索引
由字典组成的字典	各内层字典会成为一列。键会被合并成结果的行索引，跟“由Series组成的字典”的情况一样
字典或Series的列表	各项将会成为DataFrame的一行。字典键或Series索引的并集将会成为DataFrame的列标
由列表或元组组成的列表	类似于“二维ndarray”
另一个DataFrame	该DataFrame的索引将会被沿用，除非显式指定了其他索引
NumPy的MaskedArray	类似于“二维ndarray”的情况，只是掩码值在结果DataFrame会变成NA/缺失值

词典的列表生成dataframe:

If some keys in the dictionary are missing, Pandas will fill them in with `NaN` (i.e., "not a number") values:

```
data = [{ 'a': i, 'b': 2 * i }  
        for i in range(3)]
```

索引|key + 列表生成式

```
print(data)  
pd.DataFrame(data)
```

```
[{ 'a': 0, 'b': 0 }, { 'a': 1, 'b': 2 }, { 'a': 2, 'b': 4 }]
```

	a	b
0	0	0
1	1	2
2	2	4

From a two-dimensional NumPy array

Given a two-dimensional array of data, we can create a `DataFrame` with any specified column and index names. If omitted, an integer index will be used for each:

```
: 1 pd.DataFrame(np.random.rand(3, 2),  
2               columns=['foo', 'bar'],  
3               index=['a', 'b', 'c'])
```

	foo	bar
a	0.865257	0.213169
b	0.442759	0.108267
c	0.047110	0.905718

Pandas Index Object 与 表间关联

- 不可修改的数组
- 有序
- 支持可重复 key
- 表关联操作

Index计算 - 类ordered set（表关联计算的基础）

```
1 indA = pd.Index([1, 3, 5, 7, 9])
2 indB = pd.Index([2, 3, 5, 7, 11])
```

```
1 indA & indB # intersection
```

```
Int64Index([3, 5, 7], dtype='int64')
```

```
1 indA | indB # union
```

```
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
```

```
1 indA ^ indB # symmetric difference
```

```
Int64Index([1, 2, 9, 11], dtype='int64')
```

Data Selection in DataFrame (按索引选择数据)

```
1 area = pd.Series({'California': 423967, 'Texas': 695662,  
2                  'New York': 141297, 'Florida': 170312,  
3                  'Illinois': 149995})  
4 pop = pd.Series({'California': 38332521, 'Texas': 26448193,  
5                  'New York': 19651127, 'Florida': 19552860,  
6                  'Illinois': 12882135})  
7 data = pd.DataFrame({'area':area, 'pop':pop})  
8 data
```

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193



```
]:
```

```
1 data['area']
```

用法：类似多重下标

```
: California    423967
Florida        170312
Illinois       149995
New York       141297
Texas          695662
Name: area, dtype: int64
```

Equivalently, we can use attribute-style access with column names that are strings:

```
]:
```

```
1 data.area
```

用法：字段名 类比 属性

```
: California    423967
Florida        170312
Illinois       149995
New York       141297
Texas          695662
Name: area, dtype: int64
```

```
1 data['density'] = data['pop'] / data['area']  
2 data
```

	area	pop	density
California	423967	38332521	90.413926
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

行列互换操作

```
1 s = data.T
2 print(s)
3 s['California']
```

	California	Texas	New York	Florida	Illinois
area	4.239670e+05	6.956620e+05	1.412970e+05	1.703120e+05	1.499950e+05
pop	3.833252e+07	2.644819e+07	1.965113e+07	1.955286e+07	1.288214e+07
density	9.041393e+01	3.801874e+01	1.390767e+02	1.148061e+02	8.588376e+01

area 4.239670e+05

pop 3.833252e+07

density 9.041393e+01

Name: California. dtype: float64

筛选，赋值：

```
1 data.loc[data.density > 100, ['pop', 'density']]
```

	pop	density
Florida	19552860	114.806121
New York	19651127	139.076746

Any of these indexing conventions may also be used to set or modify values; this is done in the standard way that you might be accustomed to from working with NumPy:

```
1 data.iloc[0, 2] = 90
2 data
```

iloc支持多重索引

	area	pop	density
California	423967	38332521	90.000000
Texas	695662	26448193	38.018740
New York	141297	19651127	139.076746
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

Working with NumPy ufunc

```
1 df = pd.DataFrame(rng.randint(0, 10, (3, 4)),  
2                       columns=['A', 'B', 'C', 'D'])  
3 df
```

	A	B	C	D
0	9	2	6	7
1	4	3	7	7
2	2	5	4	1

```
1 np.sin(df * np.pi / 4)
```

采用Numpy的Ufunc —— 广播机制

	A	B	C	D
0	7.071068e-01	1.000000	-1.000000e+00	-0.707107
1	1.224647e-16	0.707107	-7.071068e-01	-0.707107
2	1.000000e+00	-0.707107	1.224647e-16	0.707107

Dataframe之间的运算自动进行索引对齐-补足 (out join)

Out[22]:

	A	B
0	2	4
1	18	6

► In [23]:

```
1 B = pd.DataFrame(rng.randint(0, 10, (3, 3)),  
2                       columns=list('BAC'))  
3 B
```

Out[23]:

	B	A	C
0	4	8	6
1	1	3	8
2	1	9	8

► In [24]:

```
1 A + B
```

Out[24]:

	A	B	C
0	10.0	8.0	NaN
1	21.0	7.0	NaN
2	NaN	NaN	NaN

The following table lists Python operators and their equivalent Pandas object methods:

Python Operator	Pandas Method(s)
<code>+</code>	<code>add()</code>
<code>-</code>	<code>sub()</code> , <code>subtract()</code>
<code>*</code>	<code>mul()</code> , <code>multiply()</code>
<code>/</code>	<code>truediv()</code> , <code>div()</code> , <code>divide()</code>
<code>//</code>	<code>floordiv()</code>
<code>%</code>	<code>mod()</code>
<code>**</code>	<code>pow()</code>

Frame 与 series 计算，按行broadcasting

```
: 1 A = rng.randint(10, size=(3, 4))  
2 A
```

```
: array([[9, 4, 1, 3],  
        [6, 7, 2, 0],  
        [3, 1, 7, 3]])
```

```
: 1 df = pd.DataFrame(A, columns=list('QRST'))  
2 df - df.iloc[0]
```

	Q	R	S	T
0	0	0	0	0
1	-3	3	1	-3
2	-6	-3	6	0

```
1 df.subtract(df['R'], axis=0)
```

	Q	R	S	T
0	5	0	-3	-1
1	-1	0	-5	-7
2	2	0	6	2

运算过程中类型自适应转换

The following table lists the upcasting conventions in Pandas when NA values are introduced:

Typeclass	Conversion When Storing NAs	NA Sentinel Value
floating	No change	np. nan
object	No change	None or np. nan
integer	Cast to float64	np. nan
boolean	Cast to object	None or np. nan

Keep in mind that in Pandas, string data is always stored with an `object` dtype.

Detecting null values

Pandas data structures have two useful methods for detecting null data: `isnull()` and `notnull()`. Either one will return a Boolean mask over the data. For example:

```
data = pd.Series([1, np.nan, 'hello', None])
```

```
data.isnull()
```

```
0    False
1     True
2    False
3     True
dtype: bool
```

As mentioned in [Data Indexing and Selection](#), Boolean masks can be used directly as a `Series` or `DataFrame` index:

```
data[data.notnull()]
```

```
0    1
2  hello
dtype: object
```

We can fill NA entries with a single value, such as zero:

```
data.fillna(0)
```

```
a    1.0  
b    0.0  
c    2.0  
d    0.0  
e    3.0  
dtype: float64
```

We can specify a forward-fill to propagate the previous value forward:

```
# forward-fill  
data.fillna(method='ffill')
```

```
a    1.0  
b    1.0  
c    2.0  
d    2.0  
e    3.0  
dtype: float64
```

层次-组合 索引 (Hierarchical-Indexing)

```
1 index = [('California', 2000), ('California', 2010),
2         ('New York', 2000), ('New York', 2010),
3         ('Texas', 2000), ('Texas', 2010)]
4 populations = [33871648, 37253956,
5               18976457, 19378102,
6               20851820, 25145561]
7 pop = pd.Series(populations, index=index)
8 pop
```

```
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
(Texas, 2010)         25145561
dtype: int64
```

```
# pop[:, 2000] # 这个会出错
pop[[i for i in pop.index if i[1] == 2010]]
```

```
(California, 2010)    37253956
(New York, 2010)      19378102
(Texas, 2010)         25145561
dtype: int64
```

```
index = [('California', 2000), ('California', 2010),  
         ('New York', 2000), ('New York', 2010),  
         ('Texas', 2000), ('Texas', 2010)]  
index = pd.MultiIndex.from_tuples(index) # 加个转换  
populations = [33871648, 37253956,  
               18976457, 19378102,  
               20851820, 25145561]  
pop = pd.Series(populations, index=index)  
pop[:, 2000]
```

California	33871648
New York	18976457
Texas	20851820
dtype:	int64

类似二维表切片

1	pop[:, 2010]
California	37253956
New York	19378102
Texas	25145561
dtype:	int64

MultilIndex VS extra dimension

```
1 #unstack() method will quickly convert a multiply indexed Series
2 #into a conventionally indexed DataFrame:
3 pop_df = pop.unstack() ←
4 pop_df
```

	2000	2010
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

```
1 #unstack() method will quickly convert a multiply indexed Series into a conventionally indexed DataFrame:
2 pop_df.stack()
```

California	2000	33871648
	2010	37253956
New York	2000	18976457
	2010	19378102
Texas	2000	20851820
	2010	25145561

dtype: int64


```

1 pop_df = pd.DataFrame({'total': pop,
2                        'under18': [9267089, 9284094,
3                                   4687374, 4318033,
4                                   5906301, 6879014]})
5 pop_df

```

		total	under18
California	2000	33871648	9267089
	2010	37253956	9284094
New York	2000	18976457	4687374
	2010	19378102	4318033
Texas	2000	20851820	5906301
	2010	25145561	6879014

```

1 f_u18 = pop_df['under18'] / pop_df['total']
2 f_u18.unstack()

```

	2000	2010
California	0.273594	0.249211
New York	0.247010	0.222831
Texas	0.283251	0.273568

Methods of MultiIndex Creation

The most straightforward way to construct a multiply indexed `Series` or `DataFrame` is to simply pass a list of two or more index arrays to the constructor. For example:

```
2]: 1 df = pd.DataFrame(np.random.rand(4, 2),  
2      index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],  
3      columns=['data1', 'data2'])  
4 df
```

```
]:
```

		data1	data2
a	1	0.554233	0.356072
	2	0.925244	0.219474
b	1	0.441759	0.610054
	2	0.171495	0.886688

Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a `MultiIndex` by default:

```
1 data = {('California', 2000): 33871648,  
2         ('California', 2010): 37253956,  
3         ('Texas', 2000): 20851820,  
4         ('Texas', 2010): 25145561,  
5         ('New York', 2000): 18976457,  
6         ('New York', 2010): 19378102}  
7 pd.Series(data)
```

California	2000	33871648
	2010	37253956
New York	2000	18976457
	2010	19378102
Texas	2000	20851820
	2010	25145561

dtype: int64

MultiIndex constructor

```
1 pd.MultiIndex.from_arrays(['a', 'a', 'b', 'b'], [1, 2, 1, 2])
```

```
MultiIndex(levels=['a', 'b'], [1, 2],  
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can construct it from a list of tuples giving the multiple index values of each point:

```
1 pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])
```

```
MultiIndex(levels=['a', 'b'], [1, 2],  
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can even construct it from a Cartesian product of single indices:

```
1 pd.MultiIndex.from_product(['a', 'b'], [1, 2])
```

```
MultiIndex(levels=['a', 'b'], [1, 2],  
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```



Data Aggregations on Multi-Indices

- Pandas has built-in data aggregation methods,
- such as `mean()`, `sum()`, and `max()`.
- For hierarchically indexed data, these can be passed a level parameter that controls which **subset of the data the aggregate is computed on**.
- Group by certain Key

```
1 data_mean = health_data.mean(level='year')
2 data_mean
```

subject	Bob		Guido		Sue	
type	HR	Temp	HR	Temp	HR	Temp
year						
2013	37.5	38.2	41.0	35.85	32.0	36.95
2014	38.5	37.6	43.5	37.55	56.0	36.70

```
1 health_data
```

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	31.0	38.7	32.0	36.7	35.0	37.2
	2	44.0	37.7	50.0	35.0	29.0	36.7
2014	1	30.0	37.4	39.0	37.8	61.0	36.9
	2	47.0	37.8	48.0	37.3	51.0	36.5

Combining Datasets: Merge and Join

- *one-to-one*
- *many-to-one*
- *many-to-many* joins.

Combining Datasets: Concat and Append

```
1 ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
2 ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
3 pd.concat([ser1, ser2])
```

```
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

```
1 df1 = make_df('AB', [1, 2])
2 df2 = make_df('AB', [3, 4])
3 display('df1', 'df2', 'pd.concat([df1, df2])')
```

df1			df2			pd.concat([df1, df2])		
	A	B		A	B		A	B
1	A1	B1	3	A3	B3	1	A1	B1
2	A2	B2	4	A4	B4	2	A2	B2
						3	A3	B3
						4	A4	B4

One-to-one joins:

```
1 df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
2                      'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
3 df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
4                      'hire_date': [2004, 2008, 2012, 2014]})
5 display(df1, df2)
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

```
1 df3 = pd.merge(df1, df2)
2 df3
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

Many-to-one joins 有重键值与对应的唯一键值进行join, 单值对应的记录展开为多个:

```
1 df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
2                      'supervisor': ['Carly', 'Guido', 'Steve']})
3 display('df3', 'df4', 'pd.merge(df3, df4)')
```

df3

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

df4

	group	supervisor
0	Accounting	Carly
1	Engineering	Guido
2	HR	Steve

pd.merge(df3, df4)

	employee	group	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3	Sue	HR	2014	Steve

Pandas apply方法

```
import pandas as pd

df = pd.DataFrame({'A': ['bob', 'john', 'bob', 'jeff', 'bob', 'jeff', 'bob', 'john'],
                  'B': ['one', 'one', 'two', 'three', 'two', 'two', 'one', 'three'],
                  'C': [3, 1, 4, 1, 5, 9, 2, 6],
                  'D': [1, 2, 3, 4, 5, 6, 7, 8]}) # 给出4栏数据

grouped = df.groupby('A') # 按属性A的值进行分组

for name, group in grouped:
    print(name) # 唯一的属性值
    print(group)
```

```
bob
   A  B  C  D
0  bob one  3  1
2  bob two  4  3
4  bob two  5  5
6  bob one  2  7
jeff
   A  B  C  D
3  jeff three  1  4
5  jeff two  9  6
john
   A  B  C  D
1  john one  1  2
7  john three  6  8
```

```
d = grouped.apply(lambda x: x.head(2)) # 留前两条
d
```

```
      A  B  C  D
bob 0  bob one  3  1
    2  bob two  4  3
jeff 3  jeff three  1  4
    5  jeff two  9  6
john 1  john one  1  2
    7  john three  6  8
```

自定义最大向前匹配函数

```
table['words'] = table['contance'].apply(lambda x: ' '.join(list(cut(x, word_list, 3))))|
```

table

	ID	Poem_id	line_number	contance	words
0	1	4371	-100	##饒唐永昌(一作饒唐郎中洛陽令)	饒唐永昌 一作 饒唐郎中 洛陽令
1	2	4371	-1	SS沈佺期	沈期
2	3	4371	1	洛陽舊有(一作出)神明宰	洛陽舊有 一作出 神明宰
3	4	4371	2	羣鷁由來天地中	羣鷁由來天地中
4	5	4371	3	餘邑政成何足貴	餘邑政成何足貴
...
46272	46273	39205	-1	SS李舜弦	李舜弦

1.2 统计每个词的TF-IDF值

按照空格分开, stack一下

```
split_words = table['words'].str.split(' ', expand=True).stack().rename('word').reset_index()
new_data = pd.merge(table['Poem_id'], split_words, left_index=True, right_on='level_0')
new_data
```

	Poem_id	level_0	level_1	word
0	4371	0	0	饒
1	4371	0	1	唐
2	4371	0	2	永昌
3	4371	0	3	一作
4	4371	0	4	饒
...
198498	39205	46275	4	屏



Reference :

- ▶ Python Data Science Handbook:

<https://www.oreilly.com/library/view/python-data-science/9781491912126/>