第4章 常用模块

学习目标

- 利用Numpy模块实现数组操作和运算
- 利用Pandas模块实现Series 和DataFrame数据结构的基本操作
- 利用Pandas模块实现数据统计、数据缺失、数据离散化处理
- 利用NLTK 模块实现英文文本的基本处理

4.1 Numpy

- NumPy（Numerical Python extensions）前身是1995年开始开发的一个用于数组运算的库，用于科学计算，极大地简化了向量和矩阵的操作处理。
- Numpy提供了一种新的数据结构：ndarray（n维数组，n-dimensional array），是描述相同类型的元素集合。ndarray中的每个元素是数据类型对象的对象（称为dtype），其在内存中使用相同大小的区域。
- 不同于列表和元组，数组只能存放相同数字类型的对象（如所有元素均为整型或浮点型），这使得数组上的一些运算远远快于列表上的相同运算。
- ndarray的维度（又称维数）称为秩（rank），每一个线性的数组称为一个轴（axis），也就是维度。

4.1.1 ndarray的创建

- 最简单的建立ndarray对象的方法是使用numpy.array()函数，传入一个列表来创建数组：

```
>>> import numpy as np
>>> array = np.array([2, 3, 4])
>>> print(array)
[2 3 4]
```

- 如果想要创建指定数字类型的数组，可传入指定数字类型的列表，也可通过指定ndarray的数字类型来实现：

```
>>> array1 = np.array([2., 3., 4.])
>>> array2 = np.array([2, 3, 4], dtype = float)
>>> print(array1,array2)
[2. 3. 4.] [2. 3. 4.]
```

- 利用ndim参数可以指定ndarray的维数来创建多维数组，也可以通过传入一个多维列表来创建：

```
>>> array1 = np.array([2, 3, 4], ndmin = 2)      #ndmin  指定生成数组的最小维度
>>> array2 = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(array1,array2)


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

4.1.1 ndarray的创建

- 可以通过arange()和linspace()函数创建ndarray对象，这两个函数可以创建一个一维等差数列的数组，不同之处在于arange()是以固定步长的方式创建数组，而linspace()则是以固定元素数量的方式：

```
>>> array1 = np.arange(start = 0, stop = 5.5, step = 0.5)
>>> array2 = np.linspace(start = 0, stop = 5, num = 11)
>>> print(array1, array2)
[0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ] [0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ]
```

- 还可以利用相同数值来快速创建数组，这类函数有zeros()、ones()和full()。这三种函数的不同之处在于zeros()是0填充数组，而ones()是1填充数组，full()可以使用指定的值填充数组：

```
>>> array1 = np.full(fill_value = 0, shape = (2,2))
>>> array2 = np.zeros((2,2))
>>> print(array1, array2)
[[0 0]
 [0 0]] [[0. 0.]
 [0. 0.]]
```

- 利用随机数也可以创建数组，在numpy.random子模块中有很多可以创建随机数数组的方法。
- 比较特殊的建立ndarray对象的函数是fromfunction()，该函数可以利用自定义的函数来创建数组。

np.random.rand(2,3)

4.1.2 ndarray的常用属性

- n维数组的常用属性如下：

属性	说明
ndarray.ndim	秩，即轴的数量或维度的数量
ndarray.shape	数组的形状
ndarray.size	数组元素的总个数，相当于.shape中n*m的值
ndarray.dtype	ndarray对象的元素类型
ndarray.itemsize	ndarray对象中每个元素的大小，以字节为单位

- 通过以下语句查看：

```
>>> array = np.random.rand(2, 3)
>>> print(array)
[[0.60224171 0.44621606 0.09091199]
 [0.3954423  0.66917961 0.7211016 ]]
>>> print(array.ndim)
2
>>> print(array.shape)
(2, 3)
```

```
>>> print(array.size)
6
>>> print(array.dtype)
float64
>>> print(array.itemsize)
8
```

4.1.3 ndarray的形状改变

- 可以使用reshape()或resize()方法来重新定义ndarray的形状。
- 使用reshape()并不会直接修改array本身的形状，传入reshape() 的第二个元素为-1表示自动计算列数：

```
>>> array = np.ones(shape = (12,), dtype = int)
>>> array1 = array.reshape((4, -1))
>>> print(array1)
[[1 1 1]
 [1 1 1]
 [1 1 1]
 [1 1 1]]
```

```
>>> print(array)
[1 1 1 1 1 1 1 1 1 1 1 1]
```

- 如果需要直接修改数组的形状，可以使用resize()方法：

```
>>> array.resize(4, 3)
>>> print(array)
[[1 1 1]
 [1 1 1]
 [1 1 1]
 [1 1 1]]
```

- 也可以直接修改数组的shape属性改变数组的形状。

```
>>> arr.shape = (4, 3)
```

4.1.4 ndarray的索引与切片

- ndarray的索引和切片操作和列表类似，[start : stop : step]的索引形式可用于从数组中获取片段：

```
>>> array = np.arange(5)
>>> print(array)
[0 1 2 3 4]
>>> print(array[0])
0
>>> print(array[::2])
[0 2 4]
>>> print(array[::-1])
[4 3 2 1 0]
```

4.1.4 ndarray的索引与切片

- 对于高维数组，索引和切片也有类似的操作，不同的是需要在每个维度之间用‘,’隔开：

```
>>> array = np.arange(12)
>>> array.shape = (3, 4)
>>> print(array)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> print(array[1, 2])
6
>>> print(array[:2, :2])
[[0 1]
 [4 5]]
>>> print(array[::-1, ])
[[ 8  9 10 11]
 [ 4  5  6  7]
 [ 0  1  2  3]]
```

4.1.5 ndarray的拷贝

- 列表的一个切片是它的一个拷贝，而数组的一个切片是数组上的一个视图，切片和原始数组都引用的是同一块内存区域。故当改变视图内容时，原始数组的内容也被同样改变：

```
>>> array1 = np.arange(5)
>>> array2 = array1[::2]
>>> print(array1)
[0 1 2 3 4]
>>> print(array2)
[0 2 4]
>>> array2[0] = 4
>>> print(array1)
[4 1 2 3 4]
>>> print(array2)
[4 2 4]
```

4.1.5 ndarray的拷贝

- 为了避免修改原数组，可以使用copy()来拷贝切片：

```
>>> array1 = np.arange(5)
>>> array2 = array1[::2].copy()
>>> print(array1)
[0 1 2 3 4]
>>> print(array2)
[0 2 4]
>>> array2[0] = 4
>>> print(array1)
[0 1 2 3 4]
>>> print(array2)
[4 2 4]
```

4.1.6 ndarray的拼接

- 可以使用hstack()或vstack()将多个数组拼接起来。
- hstack()是行拼接，必须保证ndarrays数组的维度相同；vstack()是列拼接，要求ndarrays数组每维长度相同：

```
>>> array1 = np.array([1, 3, 5])                    # shape = (1, 3)
>>> array2 = np.array([[2, 4, 6], [1, 5, 9]])       # shape = (2, 3)
>>> array3 = np.array([2, 4, 6, 8])                 # shape = (1,4
>>> harray = np.hstack((array1, array3))
>>> print(harray)
[1 3 5 2 4 6 8]
>>> varray = np.vstack((array1, array2))
>>> print(varray)
[[1 3 5]
 [2 4 6]
 [1 5 9]]
```

4.1.7 ndarray的运算

- ndarray对象可实现加减乘除等基本运算。既可通过基本运算符实现，也可通过NumPy自带的函数实现。
- 如‘+’操作与NumPy自带的函数add()是等价的：

```
>>> array1 = np.array([[1, 3, 5], [2, 4, 6]])
>>> array2 = np.array([[1, 2, 3], [3, 4, 5]])
>>> print(array1 + array2)
[[ 2  5  8]
 [ 5  8 11]]
>>> print(np.add(array1, array2))
[[ 2  5  8]
 [ 5  8 11]]
```

4.1.7 ndarray的运算

- 如果要实现矩阵运算，则需要使用dot()函数实现：

```
>>> array2 = np.transpose(array2)
>>> print(array2)
[[1 3]
 [2 4]
 [3 5]]
```

- NumPy还有很多高效的数学函数，如sqrt()、exp()、log()等：

```
>>> print(np.sqrt(x))
[2. 4. 8.]
>>> print(np.exp(x))
[5.45981500e+01 8.88611052e+06 6.23514908e+27]
>>> print(np.log(x))
[1.38629436 2.77258872 4.15888308]
```

4.2 Pandas

- Pandas是基于NumPy的一种工具，该工具可以解决数据分析任务。
- Pandas纳入了大量库和一些标准的数据模型，并提供了高效操作大型数据集所需的工具。
- Pandas中常用的数据结构有：
  - Series：由一组数据以及一组与之相关的数据标签（即索引）组成，类似于列表和Numpy中的一维数组。
  - DataFrame：二维表格型数据结构，含有一组有序的列，每列可以是不同的类型（数值、字符串、布尔值等），每列都有标签，可看做一个Series的字典。
  - Panel：三维数组，为DataFrame的容器。

4.2.1 Series的创建

- Series对象可以通过Series()方法创建，向该方法中传入列表、元组或字典均可。
- 当传入的是列表或元组时，若不传入index参数，默认索引为0~(N-1)的整数：

```
>>> import pandas as pd
>>> a = pd.Series(data = [5, 2.0, 's'])
>>> print(a)
0    5
1    2
2    s
dtype: object
```

- 当传入字典时，默认使用字典的键值作为Series的索引：

```
>>> dic = {'a':100, 'b': 200, 'c':300}
>>> a = pd.Series(data = dic)
>>> print(a)
a    100
b    200
c    300
dtype: int64
```

4.2.1 Series的创建

- 如果设置了index参数，则Series()方法会优先匹配所给的index参数和字典的键值，并按照index参数的顺序对传入的数据进行排序：
- 如果找不到字典中的键值和index匹配，则会在对应index标签对应的数据结果设为NaN（Not A Number，也是一个数据类型，不是空值）：

```
>>> dic = {'a':100, 'b': 200, 'c':300}
>>> a = pd.Series(data = dic, index = ['a', 'b', 'd'])
>>> print(a)
a    100.0
b    200.0
d      NaN
dtype: float64
```

4.2.2 Series的索引及切片

- 在创建Series对象时，默认Series的名称为None，可以通过设定name属性为Series命名，也可以在创建Series后通过name属性修改：
- 而索引列的名称一般通过index属性修改：

```
>>> dic = {'a':100, 'b': 200, 'c':300}
>>> a = pd.Series(data = dic)
>>> a.name = 'number'
>>> a.index.name = 'idx'
>>> print(a)
idx
a    100
b    200
c    300
Name: number, dtype: int64
```

4.2.2 Series的索引及切片

- 如果要修改Series对象中的值，和字典的修改方法类似，只需要通过索引即可访问对应的值，利用赋值操作便可将Series对象中的值进行修改：

```
>>> a['a'] = 130
>>> print(a)
idx
a    130
b    200
c    300
Name: number, dtype: int64
```

4.2.2 Series的索引及切片

- 如果需要修改索引，可使用Series.rename()方法，通过传入字典进行修改，字典的键是需要修改的索引，字典的值为新的索引：

```
>>> a = a.rename(index={'a': 'z'})
>>> print(a)
idx
z    130
b    200
c    300
Name: number, dtype: int64
```

- 如果需要修改的索引较多，则可用一个新索引列表通过赋值操作直接修改Series的索引：

```
>>> a.index = ['x', 'y', 'z']
>>> print(a)
x    130
y    200
z    300
Name: number, dtype: int64
```

4.2.3 DataFrame的创建

- DataFrame既有行索引也有列索引，可以视为由Series组成的字典（共用一个索引）。
- 最常用的创建方法是直接将一个由等长列表或NumPy数组构成的字典传入DataFrame()方法。通过以下三种输入创建的DataFrame对象是等价的：
  - 方法1：非嵌套字典

```
>>> dic1 = {'name': ['张三', '李四', '王五'],
        'age': [10, 20, 80],
        'gender': ['f', 'm', 'f']}
>>> df1 = pd.DataFrame(dic1)
>>> print(df1)
  name  age gender
0   张三   10      f
1   李四   20      m
2   王五   80      f
```

4.2.3 DataFrame的创建

- DataFrame既有行索引也有列索引，可以视为由Series组成的字典（共用一个索引）。
- 最常用的创建方法是直接将一个由等长列表或NumPy数组构成的字典传入DataFrame()方法。通过以下三种输入创建的DataFrame对象是等价的：
  - 方法2：嵌套字典

```
>>> dic2 = {'name': {0: '张三', 1:'李四', 2:'王五'},
        'age': {0: 10, 1: 20, 2: 80},
        'gender': {0: 'f', 1: 'm', 2: 'f'}}
>>> df2 = pd.DataFrame(dic2)
>>> print(df2)
  name  age gender
0   张三   10      f
1   李四   20      m
2   王五   80      f
```

4.2.3 DataFrame的创建

- DataFrame既有行索引也有列索引，可以视为由Series组成的字典（共用一个索引）。
- 最常用的创建方法是直接将一个由等长列表或NumPy数组构成的字典传入DataFrame()方法。通过以下三种输入创建的DataFrame对象是等价的：
  - 方法3：列表

```
>>> dic3 = [[10, 'f', '张三'], [20, 'm', '李四'], [80, 'f', '王五']]
>>> df3 = pd.DataFrame(dic3, columns = ['age', 'gender', 'name'])
>>> print(df3)
   age gender name
0   10      f   张三
1   20      m   李四
2   80      f   王五
```

4.2.4 DataFrame的写入与读取

- Pandas支持将DataFrame对象的读写操作，其可以支持很多联系的文件格式，最常用的是csv和xlsx格式。
- 当存在一个DataFrame时，可通过to_csv()或to_excel()函数创建一个csv或xlsx格式文件保存到当前路径：

```
>>> dic = {'name': ['Alice', 'Jack', 'Bob'],
       'age': [10, 20, 80],
       'gender': ['f', 'm', 'f']}
>>> df = pd.DataFrame(dic)
>>> df.to_csv('./person.csv')
>>> df.to_excel('./person.xlsx')
```

df1.to_csv('output/person.csv', encoding='utf_8_sig')

4.2.4 DataFrame的写入与读取

- 读取csv或xlsx格式文件时则可以使用read_csv()或read_excel ()函数：

```
>>> df1 = pd.read_csv('./person.csv')
>>> print(df1)
   Unnamed: 0   name  age gender
0           0  Alice   10      f
1           1   Jack   20      m
2           2    Bob   80      f
>>> df2 = pd.read_excel('./person.xlsx')
>>> print(df2)
    name  age gender
0  Alice   10      f
1   Jack   20      m
2    Bob   80      f
```

- 注意使用to_csv()将DataFrame对象保存为csv格式文件时会自动保存索引。当读取该文件时，会将保存的索引识别为新的一列，由于原始索引没有命名，因此以‘Unnamed: 0’进行自动命名。

4.2.5 DataFrame的索引

- DataFrame可以理解为由行和列构成的二维表格，其行索引可以使用DataFrame.index（相当于Series.name）进行查看，列索引可以使用DataFrame.columns（相当于Series.index）进行查看：

```
>>> df = pd.read_csv('./person.csv')
>>> print(df)
   Unnamed: 0   name  age gender
0           0  Alice   10      f
1           1   Jack   20      m
2           2    Bob   80      f
>>> print(df.index)
RangeIndex(start=0, stop=3, step=1)

>>> print(df.columns)
Index(['Unnamed: 0', 'name', 'age', 'gender'], dtype='object')
```

4.2.5 DataFrame的索引

- 也可以使用rename()、reset_index()和直接修改DataFrame属性等方法对索引进行修改。
- rename()只有在inplace=True时，才能在原来的DataFrame上进行修改，而reset_index()则在drop=True时丢弃原有索引，否则原有索引将新生成一列名为'index'的列：
  - 修改DataFrame的行索引：

```
>>> df.rename(index = {0: 10, 1: 20, 2: 30}, inplace = True)
>>> print(df.index)
Int64Index([10, 20, 30], dtype='int64')

>>> df.reset_index(inplace = True,drop = True)
>>> print(df.index)
RangeIndex(start=0, stop=3, step=1)
```

4.2.5 DataFrame的索引

- rename()只有在inplace=True时，才能在原来的DataFrame上进行修改，而reset_index()则在drop=True时丢弃原有索引，否则原有索引将新生成一列名为'index'的列：
  - 修改DataFrame的列索引：

```
>>> df.rename(columns = {'gender':'sex'}, inplace = True)
>>> print(df.columns)
Index(['Unnamed: 0', 'name', 'age', 'sex'], dtype='object')

>>> df.columns = ['', 'age', 'gender', 'name']
>>> print(df.columns)
Index(['', 'age', 'gender', 'name'], dtype='object')
```

- 除了上述提到的常用方法，还有set_index()、reindex()等方法能修改索引。

4.2.6 DataFrame的增删改查

- 增加
  - 通过赋值进行增加行列的操作。
  - 增加列：

```
>>> df = pd.read_csv('./person.csv')
>>> print(df)
   Unnamed: 0   name  age gender
0           0  Alice   10      f
1           1   Jack   20      m
2           2    Bob   80      f
>>> df['marriage'] = 1
>>> print(df)
   Unnamed: 0   name  age gender  marriage
0           0  Alice   10      f         1
1           1   Jack   20      m         1
2           2    Bob   80      f         1
```

4.2.6 DataFrame的增删改查

- 增加
  - 增加行：

```
>>> df.loc['3'] = {'age': 30, 'gender': 'm', 'name': 'Mary', 'marriage': 1}
>>> print(df)
   Unnamed: 0   name  age gender  marriage
0         0.0  Alice   10      f         1
1         1.0   Jack   20      m         1
2         2.0    Bob   80      f         1
3         NaN   Mary   30      m         1
>>> new = pd.Series({'age': 20, 'gender': 'm', 'name': 'Sophia', 'marriage': 0})
>>> df = df.append(new, ignore_index = True)
>>> print(df)
   Unnamed: 0    name  age gender  marriage
0         0.0   Alice   10      f         1
1         1.0    Jack   20      m         1
2         2.0     Bob   80      f         1
3         NaN    Mary   30      m         1
4         NaN  Sophia   20      m         0
```

4.2.6 DataFrame的增删改查

- 查询
  - 通过loc和iloc可查看某些位置的具体数值，也可以通过指定行索引或列索引查看整行或整列的结果。
  - loc为按标签查找，而iloc是按位置查找，位置从0开始标记，两者行和列之间均用‘,’分隔，如果需要选定多行或多列，则需要传入待选定行列表或列列表。
  - 查看具体数值：

```
>>> print(df.loc[0, 'age'])
10
>>> print(df.iloc[0, 0])
0.0
>>> print(df[df['name'] == 'Alice']['age'])
0    10
Name: age, dtype: int64
```

4.2.6 DataFrame的增删改查

- 查询
  - 查看行：

```
>>> print(df[0:1])
   Unnamed: 0   name  age gender  marriage
0         0.0  Alice   10      f         1
>>> print(df.loc[0])
Unnamed: 0         0
name           Alice
age               10
gender             f
marriage           1
Name: 0, dtype: object
```

4.2.6 DataFrame的增删改查

- 查询
  - 查看列：

```
>>> print(df['name'])
0     Alice
1      Jack
2       Bob
3      Mary
4    Sophia
Name: name, dtype: object
>>> print(df.loc[:,'name'])
0     Alice
1      Jack
2       Bob
3      Mary
4    Sophia
Name: name, dtype: object
```

  - 除了loc和iloc外，还有ix等方式可以进行查询。

4.2.6 DataFrame的增删改查

- 修改
  - loc和iloc得到的是DataFrame的一个视图，故可使用查询进行定位，再通过赋值操作对DataFrame中的值进行修改：
  - 修改整列：

```
>>> df['gender'] = 'f'
>>> print(df)
   Unnamed: 0    name  age gender  marriage
0         0.0   Alice   10      f         1
1         1.0    Jack   20      f         1
2         2.0     Bob   80      f         1
3         NaN    Mary   30      f         1
4         NaN  Sophia   20      f         0
```

4.2.6 DataFrame的增删改查

- 修改
  - 修改指定位置：

```
>>> df.iloc[1,3] = 'm'
>>> df.loc[0, 'marriage'] = 0
>>> print(df)
   Unnamed: 0    name age gender  marriage
0         0.0   Alice  10      f         0
1         1.0    Jack  20      m         1
2         2.0     Bob  80      f         1
3         NaN    Mary  30      f         1
4         NaN  Sophia  20      f         0
```

4.2.6 DataFrame的增删改查

- 删除
  - 删除操作可以使用drop()方法，需要通过指定行索引或列索引删除某列或某行，也可以通过传入一个待删除的索引列表，指明其所在的轴axis来进行多行或多列的删除，（0表示删除行，1表示删除列）。
  - inplace=True时才能直接对原来的DataFrame进行修改。
  - 删除行：

```
>>> df.drop(0, axis = 0, inplace = True)
>>> print(df)
   Unnamed: 0    name age gender  marriage
1         1.0    Jack  20      m         1
2         2.0     Bob  80      f         1
3         NaN    Mary  30      f         1
4         NaN  Sophia  20      f         0
```

4.2.6 DataFrame的增删改查

- 删除
  - 删除列：

```
>>> df.drop(['gender', 'marriage'], axis = 1, inplace = True)
>>> print(df)
   Unnamed: 0    name age
1         1.0    Jack  20
2         2.0     Bob  80
3         NaN    Mary  30
4         NaN  Sophia  20
```

  - DataFrame的删除操作也可以通过pop()、del等进行。

4.2.7 DataFrame的数据统计方法

- 使用部分函数对加利福尼亚住房数据集进行初步探索：

```
>>> data = pd.read_csv('./cal_housing.data', header = None)
>>> data.columns = ['longitude', 'latitude', 'housingMedianAge', 'totalRooms', 'tot
alBedrooms', 'population', 'households', 'medianIncome', 'medianHouseValue']
```

- 查看形状：

```
>>> print(data.shape)
(20640, 9)
```

- 查看前5行：

```
>>> print(data.head())
   longitude  latitude  ...  medianIncome  medianHouseValue
0    -122.23     37.88  ...        8.3252          452600.0
1    -122.22     37.86  ...        8.3014          358500.0
2    -122.24     37.85  ...        7.2574          352100.0
3    -122.25     37.85  ...        5.6431          341300.0
4    -122.25     37.85  ...        3.8462          342200.0

[5 rows x 9 columns]
```

4.2.7 DataFrame的数据统计方法

- 使用部分函数对加利福尼亚住房数据集进行初步探索：
  - 按行进行描述性统计：

```
>>> print(data.describe())
          longitude      latitude  ...  medianIncome  medianHouseValue
count  20640.000000  20640.000000  ...  20640.000000      20640.000000
mean    -119.569704     35.631861  ...      3.870671     206855.816909
std        2.003532      2.135952  ...      1.899822     115395.615874
min     -124.350000     32.540000  ...      0.499900      14999.000000
25%     -121.800000     33.930000  ...      2.563400     119600.000000
50%     -118.490000     34.260000  ...      3.534800     179700.000000
75%     -118.010000     37.710000  ...      4.743250     264725.000000
max     -114.310000     41.950000  ...     15.000100     500001.000000

[8 rows x 9 columns]
```

4.2.7 DataFrame的数据统计方法

- 使用部分函数对加利福尼亚住房数据集进行初步探索：
  - 分组统计：

```
>>> gb = data.groupby('housingMedianAge')
>>> print(gb['totalRooms'].max())
housingMedianAge
1.0       2254.0
2.0      30450.0
3.0      39320.0
4.0      37937.0
5.0      27870.0
6.0      24121.0
7.0      28258.0
8.0      32054.0
9.0      30405.0
10.0     20263.0
```

4.2.8 缺失数据处理

- DataFrame有三种常用的方法来查看和处理数据缺失：isnull()方法、dropna()方法和fillna()方法。
- isnull()方法：
  - 该方法可以判断DataFrame中哪些是缺失值：

```
>>> dic = {'name': ['张三', '李四', '王五'], 'gender': ['f', 'm', 'f']}
>>> df = pd.DataFrame(dic, columns=['name', 'gender', 'age'])
>>> print(df)
  name gender  age
0   张三      f  NaN
1   李四      m  NaN
2   王五      f  NaN
>>> print(df.isnull())
    name  gender   age
0  False   False  True
1  False   False  True
2  False   False  True
```

4.2.8 缺失数据处理

- dropna()方法：
  - 该方法可以删除缺失值，常用的参数有axis、how和subset三个参数。axis是指定处理的轴，0表示行，1表示列；how是指定该如何删除缺失值，‘all’表示行或列全为缺失值才删除，‘any’表示行或者列只要有缺失值就删除：

```
>>> df1 = df.dropna(axis = 1, how = 'all')
>>> print(df1)
  name gender
0   张三      f
1   李四      m
2   王五      f
```

- fillna()方法：
  - 该方法可以用指定的值对缺失值进行填充。该方法与dropa()方法需要指定inplace=True，否则不会对原有DataFrame进行修改：

```
>>> df2 = df.fillna(value = 20)
>>> print(df2)
  name gender  age
0   张三      f   20
1   李四      m   20
2   王五      f   20
```

4.2.9 数据离散化

- 数据离散化是将取值连续的属性转化为分类属性。DataFrame支持的离散化方法有cut()方法和qcut()方法：
- cut()方法：
  - 即等宽法，将属性的值域分成具有相同宽度的区间，参数包括待分组的数据（x）和组数（bins）：

```
>>> data = np.random.randn(100)
>>> group1 = pd.cut(data, bins = 5)
>>> print('group1:\n', group1.value_counts())
group1:
 (-2.141, -1.197]     8
(-1.197, -0.259]    24
(-0.259, 0.68]      43
(0.68, 1.619]       21
(1.619, 2.557]       4
dtype: int64
```

4.2.9 数据离散化

- qcut()方法：
  - 即等频法，按照相同的频数将属性分成不同的区间，参数包括待分组的数据（x）和组数（q）：

```
>>> group2 = pd.qcut(data, q = 5)
>>> print('group2:\n', group2.value_counts())
group2:
 (-2.137, -0.607]    20
(-0.607, -0.065]    20
(-0.065, 0.342]     20
(0.342, 0.728]      20
(0.728, 2.557]      20
dtype: int64
```

4.3 NLTK

- NLTK（Natural Language Toolkit） 是自然语言处理工具包，集成了大量语料库和词汇资源，提供了丰富的主要基于英文的文本处理方法。
- NLTK中常用的英文文本预处理方法包括分句与分词、词性标注、符号和停用词处理、词干提取与词形还原和词相似度计算等。

4.3.1 分句与分词

- 分句是指把由多个句子组成的文档拆分成以句子，分词则是把由多个词语组成的句子拆分成词语。
- NLTK的sent_tokenize()和word_tokenize()函数能实现分句与分词：

```
>>> import nltk
>>> doc = 'Business analytics is a field that drives practical, data-driven changes
in a business. It is a practical application of statistical analysis that focuses o
n providing actionable recommendations. Analysts in this field focus on how to appl
y the insights they derive from data.'
>>> sent = nltk.sent_tokenize(doc)
>>> print(sent)
['Business analytics is a field that drives practical, data-driven changes in a bus
iness.', 'It is a practical application of statistical analysis that focuses on pro
viding actionable recommendations.', 'Analysts in this field focus on how to apply
the insights they derive from data.']
>>> word = nltk.word_tokenize(sent[0])
>>> print(word)
['Business', 'analytics', 'is', 'a', 'field', 'that', 'drives', 'practical', ',', '
data-driven', 'changes', 'in', 'a', 'business', '.']
```

4.3.2 词性标注

- 词性标注是赋予句子中每个词准确的词性标签，如动词、名词、形容词等。
- NLTK的pos_tag()函数能进行词性标注：

```
>>> postag = nltk.pos_tag(word)
>>> print(postag)
[('Business', 'NN'), ('analytics', 'NNS'), ('is', 'VBZ'), ('a', 'DT'), ('field', 'N
N'), ('that', 'WDT'), ('drives', 'VBZ'), ('practical', 'JJ'), (',', ','), ('data-dr
iven', 'JJ'), ('changes', 'NNS'), ('in', 'IN'), ('a', 'DT'), ('business', 'NN'), ('
.', '.')]
```

4.3.3 符号和停用词处理

- 停用词是指在语言表达中常常出现但没有太多意义，通常可以忽略的词汇。英文表达中，冠词、介词以及连词等都属于停用词。
- 可以自定义停用词表或采用NLTK中自带的英文停用词表进行处理：

```
>>> from nltk.corpus import stopwords
>>> punctuation = '!"#$%&\'()*+,-./:;<=>?@[\]^_`{|}~'
>>> stw = stopwords.words('english')
>>> new = [w.lower() for w in word if w not in punctuation and w not in stw]
>>> print(new)
['business', 'analytics', 'field', 'drives', 'practical', 'data-driven', 'changes',
'business']
```

4.3.4 词干提取与词形还原

- 词干提取（Stemming）是一个将词语简化为词干、词根或词形的过程。而词形还原（Lemmatization）是将单词的不同形式还原到一个常见的基础形式。
- 词干提取往往是简单地去掉单词的前后缀，得到词根，而词形还原并不是简单地对单词进行切断或变形，而是通过使用词汇知识库来将单词的复杂形态转变成最基础的形态。
- NLTK的PorterStemmer模块能进行词干提取：

```
>>> from nltk.stem import PorterStemmer
>>> word = ['driven', 'drives', 'having', 'has', 'are', 'were']
>>> stemmer = PorterStemmer()
>>> stem_result = [stemmer.stem(w) for w in word]
>>> print(stem_result)
['driven', 'drives', 'have', 'ha', 'are', 'were']
```

4.3.4 词干提取与词形还原

- NLTK的WordNetLemmatizer模块能进行词形还原：

```
>>> from nltk.stem.wordnet import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> lem_result1 = [lemmatizer.lemmatize(w) for w in word]
>>> lem_result2 = [lemmatizer.lemmatize(w, pos = 'v') for w in word]
>>> print(lem_result1)
['driven', 'drive', 'having', 'ha', 'are', 'were']
>>> print(lem_result2)
['drive', 'drive', 'have', 'have', 'be', 'be']
```

4.3.5 词相似度计算

- NLTK中集成了多种词典，我们可以使用WordNet 计算词与词之间的相似度。
- 通过wordnet.synsets()可获得该词在WordNet中的同义词集合：

```
>>> from nltk.corpus import wordnet
>>> print(wordnet.synsets('football'))
[Synset('football.n.01'), Synset('football.n.02')]
>>> print('No.1:', wordnet.synset('football.n.01').definition())
No.1: any of various games played with a ball (round or oval) in which two teams tr
y to kick or carry or propel the ball into each other's goal
>>> print('No.2:', wordnet.synset('football.n.02').definition())
No.2: the inflated oblong ball used in playing American football
```

4.3.5 词相似度计算

- 通过wordnet.path_similarity()基于两个词在WordNet中的最短路径计算相似度：

```
>>> w1 = wordnet.synset('basketball.n.01')
>>> w2 = wordnet.synset('soccer.n.01')
>>> w3 = wordnet.synset('football.n.01')
>>> print(wordnet.path_similarity(w1, w2))
0.14285714285714285
>>> print(wordnet.path_similarity(w2, w3))
0.5
```

- 此外还有wordnet.lch_similarity()和wordnet.wup_similarity()等方法可以计算词相似度。