目录

1、numpy

因为 pandas 是基于 numpy 的，所以在学习 Pandas 之前我决定先学习一下 numpy。

1.1、对象的建立

np.ndarray 只是一个便捷的函数，可以用来创建 array

```
import numpy as np



s = np.array([1, 2, 3])
s1 = np.asarray(s)
s2 = np.zeros([4, 5])
s3 = np.zeros_like(s1)
print("s1:", s1)
print("创建全零array:", s2)
print("根据s1形状创建全零array:", s3)
```

```
s1: [1 2 3]
创建全零array: [[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
根据s1形状创建全零array: [0 0 0]
```

1.2、数组属性

shape 反映 ndarray 的大小

```
>>> a = np.random.randint(0,6,(4,4))
>>> a
array([[0, 1, 0, 5],
       [2, 0, 3, 5],
       [3, 5, 5, 1],
       [3, 2, 0, 2]])
>>> a.shape
(4, 4)
```

可以在创建 array 后，指定 shape，也可使用 reshape

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

```
>>> c = b.reshape(4,3)
>>> c
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

返回的为 b 的引用
ndim，返回 array 的维度

```
>>> b.ndim
2
```

1.3、切片与索引

array 切片与 list 切片很像，遵从[start:end:step]，区别在于 array 逗号可以区分维度。

```
>>> import numpy as np
>>> a1 = np.random.randint(1, 10, (4,5))
>>> a1
array([[5, 5, 2, 8, 5],
       [1, 3, 4, 5, 8],
       [4, 1, 2, 5, 7],
       [5, 1, 8, 1, 2]])
>>> a2 = a1[0:2:1, 0:1:1]
>>> a2
array([[5],
       [1]])
>>> a1[0:2:1]
array([[5, 5, 2, 8, 5],
       [1, 3, 4, 5, 8]])
>>> a1[3][3]
1
>>> a1[3,3]
1
```

神奇的布尔索引

```
>>> languages = np.array(['c','perl','python','c','python','perl','java'])
>>> mask = (languages == 'c')|(languages == 'java')
>>> a1 = np.arange(1,8)
>>> a2 = a1[mask]
>>> mask
array([ True, False, False,  True, False, False,  True])
>>> a1
array([1, 2, 3, 4, 5, 6, 7])
>>> a2
array([1, 4, 7])
```

1.4、array 之间的运算

矩阵乘法

```
>>> a1 = np.random.randint(3,7,[3,5])
>>> a2 = np.random.randint(3,7,[5,6])
>>> np.dot(a1,a2)
array([[ 95,  100,   96,   75,   77,   68],
       [119,  123,  111,  102,  104,   81],
       [100,  108,   97,   90,   88,   77]])
```

数乘

```
>>> a3 = np.random.randint(3,8,[3,5])
>>> a1*a3
array([[15,  15,  12,  15,  20],
       [18,  15,  36,  12,  20],
       [12,  21,  25,  18,  16]])
>>> a3*3
array([[15,   9,  12,  15,  12],
       [ 9,  15,  18,  12,  12],
       [12,  21,  15,   9,  12]])
```

1.5、广播机制

```
>>> a = np.random.randint(0,9,[4,4])
>>> a
array([[8,  7,  7,  1],
       [6,  1,  1,  6],
       [8,  2,  6,  4],
       [3,  0,  8,  3]])
>>> b = np.random.randint(2,5,[1,4])
>>> b
array([[2,  4,  3,  2]])
>>> a+b
array([[10,  11,  10,   3],
       [ 8,   5,   4,   8],
       [10,   6,   9,   6],
       [ 5,   4,  11,   5]])
```

1.6、统计函数

np.amin()、np.amax()：从给定数组中的元素沿指定轴返回最小值和最大值。
np.ptp()：函数返回沿轴的值的范围(最大值 - 最小值)。

np.mean()：算术平均值是沿轴的元素的总和除以元素的数量。 numpy.mean()函数返回数组中元素的算术平均值。 如果提供了轴，则沿其计算。

```
>>> a
array([[8, 7, 7, 1],
       [6, 1, 1, 6],
       [8, 2, 6, 4],
       [3, 0, 8, 3]])
>>> np.amin(a,0)
array([3, 0, 1, 1])
>>> np.amax(a,0)
array([8, 7, 8, 6])
>>> np.ptp(a,0)
array([5, 7, 7, 5])
>>> np.mean(a, 0)
array([6.25, 2.5 , 5.5 , 3.5 ])
```

1.7、IO 文件操作

save 和 load

```
>>> np.save('outfile', a)
>>> e = np.load('outfile.npy')
>>> e
array([[  0,   1,   0,   5],
       [  2,   0,   3,   5],
       [  3,   5, 100,   1],
       [  3,   2,   0,   2]])
```

savetxt 和 loadtxt

```
>>> np.savetxt('outfile2', a,fmt='%d', delimiter=',',newline='\n',header='start',footer='end',encoding='utf-8')
```

保存后文件

```
1  # start
2  0,1,0,5
3  2,0,3,5
4  3,5,100,1
5  3,2,0,2
6  # end
7
```

numpy.loadtxt(fname,dtype=,comments='#',delimiter=None,converters=None, skiprows=0,usecols=None, unpack=False, ndmin=0)

skiprows 为跳过 n 行，usecols 为要使用哪几列

```
>>> data2 = np.loadtxt('outfile2',delimiter=',')
>>> data2
array([[  0.,   1.,   0.,   5.],
       [  2.,   0.,   3.,   5.],
       [  3.,   5., 100.,   1.],
       [  3.,   2.,   0.,   2.]])
```

会自动忽略 header 和 footer

2、pandas 学习

2.1、pandas 基本数据结构

2.1.1、Series

像 ndarray 一样

Series 对象的创建可以使用一组序列，dict、an ndarray or scalar value。如果不设置索引会创建默认索引

```
In [1]: import pandas as pd

In [2]: import numpy as np

In [3]: a = pd.Series([1,2,4,5])

In [4]: a
Out[4]:
0    1
1    2
2    4
3    5
dtype: int64

In [5]: pd.Series(np.random.randn(5),index=['a','b','c','d','e'])
Out[5]:
a   -1.410104
b   -1.084952
c   -0.773578
d    1.006689
e    1.176528
dtype: float64
```

比较奇特的索引,传入一个列表，返回列表里面的索引值。

```
In [7]: b
Out[7]:
a    0.030339
b    1.651184
c    1.631957
d   -0.474488
e    0.010504
dtype: float64

In [8]: b[[3,4]]
Out[8]:
d   -0.474488
e    0.010504
dtype: float64
```

像 dict 一样

根据 index 直接获得值，就像 dict 的 key 一样，可以查找或是修改。

如果 index 不存在，则会报错。也可以使用 get 方法，如果 index 不存在则返回 None。与 dict 不一样，index 可以重复。

```
In [10]: b['b']
Out[10]: 1.6511842365734302

In [11]: b['b'] = 12

In [12]: b['b']
Out[12]: 12.0

In [13]: b.get('b')
Out[13]: 12.0

In [14]: b.get('t')
```

Series 与 Series 和数之间可以进行基本运算，和 ndarray 不同的是，由于 Series 含有 index，如果两个 Series 相加，倘若 index 不对应的话，则会出现数据丢失现象。可以使用 dropna 函数来舍弃不匹配的元素。

```
In [17]: c
Out[17]:
a   -1.156083
b   -1.254722
c   -1.777216
d   -1.188846
f    1.684417
dtype: float64

In [18]: b
Out[18]:
a     0.030339
b    12.000000
c     1.631957
d    -0.474488
e     0.010504
dtype: float64

In [19]: b*2
Out[19]:
a     0.060678
b    24.000000
c     3.263913
d    -0.948976
e     0.021008
dtype: float64
```

```
In [20]: b+c
Out[20]:
a     -1.125744
b     10.745278
c     -0.145259
d     -1.663334
e           NaN
f           NaN
dtype: float64

In [21]: (b+c).dropna()
Out[21]:
a     -1.125744
b     10.745278
c     -0.145259
d     -1.663334
dtype: float64
```

Name 属性

使用 rename 可以直接产生一个原 Serious 的副本。

```
In [23]: s = pd.Series(np.random.randn(5),name='first')

In [24]: s
Out[24]:
0    0.598224
1    0.507860
2   -0.214100
3   -0.357330
4   -0.647952
Name: first, dtype: float64

In [25]: s2 = s.rename('second')

In [26]: s2
Out[26]:
0    0.598224
1    0.507860
2   -0.214100
3   -0.357330
4   -0.647952
Name: second, dtype: float64
```

基本属性及方法

编号	属性或方法	描述
1	axes	返回行轴标签列表。
2	dtype	返回对象的数据类型(dtype)。
3	empty	如果系列为空，则返回 True。
4	ndim	返回底层数据的维数，默认定义：1。
5	size	返回基础数据中的元素数。
6	values	将系列作为 ndarray 返回。
7	head()	返回前 n 行。
8	tail()	返回最后 n 行。

```
In [34]: s
Out[34]:
a    6
b    0
c    1
dtype: int64

In [35]: s.axes
Out[35]: [Index(['a', 'b', 'c'], dtype='object')]

In [36]: s.dtype
Out[36]: dtype('int64')

In [37]: s.empty
Out[37]: False

In [38]: s.size
Out[38]: 3

In [39]: s.values
Out[39]: array([6, 0, 1], dtype=int64)

In [40]: s.head(2)
Out[40]:
a    6
b    0
dtype: int64

In [41]: s.tail(2)
Out[41]:
b    0
c    1
dtype: int64
```

2.1.2、DataFrame

可以通过如下数据类型建立：
Dict of 1D ndarrays, lists, dicts, or Series
2-D numpy.ndarray
A Series
Structured or record ndarray
Another DataFrame
在建立时可以指定索引，pandas 会根据索引来排序
字典建立，使用二级字典

```
In [44]: s = {'one':{'a':1,'b':2,'c':3,'d':4,'e':5},'two':{'a':11,'b':12,'c':13,'d':14,'e':15}}

In [45]: s
Out[45]:
{'one': {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5},
 'two': {'a': 11, 'b': 12, 'c': 13, 'd': 14, 'e': 15}}
```

```
In [46]: df = pd.DataFrame(s)

In [47]: df
Out[47]:
   one  two
a    1   11
b    2   12
c    3   13
d    4   14
e    5   15

In [48]: df = pd.DataFrame(s,index=['a','b'])

In [49]: df
Out[49]:
   one  two
a    1   11
b    2   12
```

通过纯 dict 建立时，外层 key 会被当作 columns，内层被当作 index
可以为 columns 和 row 分别命名，值可以使用 values 函数获得

```
In [50]: df.index
Out[50]: Index(['a', 'b'], dtype='object')

In [51]: df.index.name

In [52]: df.index.name = 'row'

In [53]: df.index.name
Out[53]: 'row'

In [54]: df.values
Out[54]:
array([[ 1, 11],
       [ 2, 12]], dtype=int64)

In [55]: df
Out[55]:
     one  two
row
a      1   11
b      2   12

In [56]: df.columns.name = 'columns'

In [57]: df
Out[57]:
columns  one  two
row
a          1   11
b          2   12
```

索引
通过位置直接索引

```
In [61]: df
Out[61]:
columns  one  two
row
a          1   11
b          2   12

In [62]: df.iat[1,1]
Out[62]: 12
```

索引某行

```
In [63]: df.iloc[1]
Out[63]:
columns
one     2
two    12
Name: b, dtype: int64

In [64]: df.loc['b']
Out[64]:
columns
one     2
two    12
Name: b, dtype: int64
```

索引某列

```
In [65]: df['one']
Out[65]:
row
a    1
b    2
Name: one, dtype: int64

In [66]: df.one
Out[66]:
row
a    1
b    2
Name: one, dtype: int64
```

修改数据

通过索引来实现修改

```
In [70]: df
Out[70]:
   one  two
a    1   11
b    2   12
c    3   13
d    4   14
e    5   15
```

```
In [72]: df['three'] = pd.Series(np.random.randint(10,18,
[5]),index=['a','b','c','d','e'])

In [73]: df
Out[73]:
   one  two  three
a    1   11     17
b    2   12     11
c    3   13     10
d    4   14     15
e    5   15     14
```

删除数据

使用 del 命令

尝试了一下，无法使用 del df.three

```
In [77]: del df['three']

In [78]: df
Out[78]:
   one  two
a    1   11
b    2   12
c    3   13
d    4   14
e    5   15
```

使用 drop 函数，参数 axis

```
In [80]: df.drop('one',axis=1)
Out[80]:
   two
a   11
b   12
c   13
d   14
e   15
```

2.2、pandas 基本功能

2.2.1、描述性统计

编号	函数	描述
1	count()	非空观测数量
2	sum()	所有值之和
3	mean()	所有值的平均值
4	median()	所有值的中位数
5	mode()	值的模值
6	std()	值的标准偏差
7	min()	所有值中的最小值
8	max()	所有值中的最大值

9	abs()	绝对值
10	prod()	数组元素的乘积
11	cumsum()	累计总和
12	cumprob()	累计乘积

函数使用方法大同小异，这里只演示 sum 函数

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Minsu','Jack',
   'Lee','David','Gasper','Betina','Andres']),
   'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65]
df = pd.DataFrame(d)
print(df)
print('sum')
print('axis=0')
print(df.sum(0))
print('axis=1')
print(df.sum(1))
```

```
    Age    Name  Rating
0    25     Tom    4.23
1    26   James    3.24
2    25   Ricky    3.98
3    23     Vin    2.56
4    30   Steve    3.20
5    29   Minsu    4.60
6    23    Jack    3.80
7    34     Lee    3.78
8    40   David    2.98
9    30  Gasper    4.80
10   51  Betina    4.10
11   46  Andres    3.65
sum
axis=0
Age                                                    382
Name      TomJamesRickyVinSteveMinsuJackLeeDavidGasperBe...
Rating                                                44.92
dtype: object
axis=1
0     29.23
1     29.24
2     28.98
3     25.56
4     33.20
5     33.60
6     26.80
7     37.78
8     42.98
9     34.80
10    55.10
11    49.65
dtype: float64
```

这里要提一下 describe 函数，它是用来计算有关 DataFrame 列的统计信息的摘要。参数列表里的 include 有三个可选项，object-汇总字符串列，number-汇总数字列，all-所有列汇总。

```
              Age      Rating
count   12.000000   12.000000
mean    31.833333    3.743333
std      9.232682    0.661628
min     23.000000    2.560000
25%     25.000000    3.230000
50%     29.500000    3.790000
75%     35.500000    4.132500
max     51.000000    4.800000
```

2.2.2、一些基本函数应用

pipe()通过自己定义的函数，将其作用到 Series 或者是 DataFrame 对象每一个元素上。

```
133 def adder(ele1, ele2):
134     return ele1+ele2
135 
136 df = pd.DataFrame(np.random.randn(5,4))
137 print(df)
138 print(df.pipe(adder,2))
```

```
          0         1         2         3
0  0.330948  0.551112  0.650601 -0.526827
1  0.533744  0.862535  1.848636  0.618539
2 -1.639617 -1.385512  0.433603 -0.945190
3 -1.051150  0.501423 -0.550315 -0.864475
4  0.313498 -0.325348 -1.041003 -1.240097
          0         1         2         3
0  2.330948  2.551112  2.650601  1.473173
1  2.533744  2.862535  3.848636  2.618539
2  0.360383  0.614488  2.433603  1.054810
3  0.948850  2.501423  1.449685  1.135525
4  2.313498  1.674652  0.958997  0.759903
```

apply()智能函数，沿 DataFrame 的轴应用任意函数，它与描述性统计方法一样，采用可选的轴参数。 默认情况下，操作按列执行，将每列列为数组。

```
147 print(df)
148 print('axis=0')
149 print(df.apply(np.sum,0))
150 print('axis=1')
151 print(df.apply(np.sum,1))
```

```
   0  1  2
0  1  2  3
1  3  4  5
axis=0
0    4
1    6
2    8
dtype: int64
axis=1
0     6
1    12
dtype: int64
```

并不是所有的函数都可以向量化(也不是返回另一个数组的 NumPy 数组，也不是任何值)，在 DataFrame 上的方法 applymap()和类似地在 Series 上的 map()

接受任何 Python 函数，并且返回单个值。个人觉得和 pipe()函数类似。

```
147 print(df)
148 print('applymap')
149 print(df.applymap(lambda x: x+3))
150 print('pipe')
151 print(df.pipe(lambda x: x+3))
```

```
   0  1  2
0  1  2  3
1  3  4  5
applymap
   0  1  2
0  4  5  6
1  6  7  8
pipe
   0  1  2
0  4  5  6
1  6  7  8
```

拼接函数 concat()

```
In [83]: df = pd.DataFrame(np.random.randn(10,4))

In [84]: pieces = [df[0:3],df[3:7],df[7:]]

In [85]: pieces
Out[85]:
[          0         1         2         3
 0  0.709041 -0.940502 -1.092342  0.254073
 1  0.387155 -0.408790 -0.724966  0.640323
 2 -0.258586 -0.370508 -0.197107  2.489258,
           0         1         2         3
 3 -0.858140 -0.263199  1.335684  0.016282
 4  1.404474 -0.775464  0.850675  1.729575
 5 -0.603999  0.610288  0.928558  1.175337
 6 -0.038709  1.333058  0.539189  1.846494,
           0         1         2         3
 7  1.765145  0.521517 -2.006910  0.310524
 8 -1.062125 -0.893208  0.057335 -0.529414
 9  1.856647  0.512445  0.162575 -2.092599]
```

```
In [87]: pd.concat(pieces)
Out[87]:
          0         1         2         3
0  0.709041 -0.940502 -1.092342  0.254073
1  0.387155 -0.408790 -0.724966  0.640323
2 -0.258586 -0.370508 -0.197107  2.489258
3 -0.858140 -0.263199  1.335684  0.016282
4  1.404474 -0.775464  0.850675  1.729575
5 -0.603999  0.610288  0.928558  1.175337
6 -0.038709  1.333058  0.539189  1.846494
7  1.765145  0.521517 -2.006910  0.310524
8 -1.062125 -0.893208  0.057335 -0.529414
9  1.856647  0.512445  0.162575 -2.092599
```

轴向连接 concatenate()通过 axis 参数来指明方向

```
In [107]: a
Out[107]:
   0  1  2  3
0  6  6  0  5
1  4  2  5  4
2  5  6  4  1
3  7  1  7  4

In [108]: np.concatenate([a,a], axis=0)
Out[108]:
array([[6, 6, 0, 5],
       [4, 2, 5, 4],
       [5, 6, 4, 1],
       [7, 1, 7, 4],
       [6, 6, 0, 5],
       [4, 2, 5, 4],
       [5, 6, 4, 1],
       [7, 1, 7, 4]])

In [109]: np.concatenate([a,a], axis=1)
Out[109]:
array([[6, 6, 0, 5, 6, 6, 0, 5],
       [4, 2, 5, 4, 4, 2, 5, 4],
       [5, 6, 4, 1, 5, 6, 4, 1],
       [7, 1, 7, 4, 7, 1, 7, 4]])
```

连接函数 jion()，类似于 sql 语句

```
In [88]: left = pd.DataFrame({'key':['foo','foo'],'lval':[1,2]})
```

```
In [90]: left
Out[90]:
   key  lval
0  foo     1
1  foo     2
```

```
In [93]: right = pd.DataFrame({'key':['foo','foo'],'rval':[4,5]})

In [94]: right
Out[94]:
   key  rval
0  foo     4
1  foo     5

In [95]: pd.merge(left,right,on='key')
Out[95]:
   key  lval  rval
0  foo     1     4
1  foo     1     5
2  foo     2     4
3  foo     2     5
```

转置

```
In [99]: c
Out[99]:
   key  lval  rval
0  foo     1     4
1  foo     1     5
2  foo     2     4
3  foo     2     5

In [100]: c.T
Out[100]:
        0    1    2    3
key   foo  foo  foo  foo
lval    1    1    2    2
rval    4    5    4    5
```

可以像 ndarray 一样进行矩阵的运算

```
In [101]: a = pd.DataFrame(np.random.randint(0,8,[4,4]))

In [102]: a
Out[102]:
   0  1  2  3
0  6  6  0  5
1  4  2  5  4
2  5  6  4  1
3  7  1  7  4

In [103]: a.dot(a)
Out[103]:
     0   1   2   3
0   95  53  65  74
1   85  62  58  49
2   81  67  53  57
3  109  90  61  62

In [104]: a+a
Out[104]:
    0   1   2   3
0  12  12   0  10
1   8   4  10   8
2  10  12   8   2
3  14   2  14   8
```

2.3、高阶应用

2.3.1、时间 series

Time series 具有简单，强大且高效的功能，可在频率转换过程中执行重采样操作（例如，将数据转换为 5 分钟的数据）。这在金融应用中非常普遍，但不限于此。

```
182 rng = pd.date_range('1/1/2012', periods=5, freq='S')
183 ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)
184 print(ts)
185
186 print(ts.resample('5Min').sum())
```

```
2012-01-01 00:00:00    156
2012-01-01 00:00:01    108
2012-01-01 00:00:02    293
2012-01-01 00:00:03     42
2012-01-01 00:00:04    176
Freq: S, dtype: int32
2012-01-01    775
Freq: 5T, dtype: int32
```

时区表示

```
rng = pd.date_range('3/6/2012 00:00', periods=5, freq='D')
ts = pd.Series(np.random.randn(len(rng)), rng)
print(ts)
ts_utc = ts.tz_localize('UTC')
print(ts_utc)
```

```
2012-03-06   -1.205229
2012-03-07    1.060393
2012-03-08    1.225148
2012-03-09   -0.388118
2012-03-10   -0.507513
Freq: D, dtype: float64
2012-03-06 00:00:00+00:00   -1.205229
2012-03-07 00:00:00+00:00    1.060393
2012-03-08 00:00:00+00:00    1.225148
2012-03-09 00:00:00+00:00   -0.388118
2012-03-10 00:00:00+00:00   -0.507513
Freq: D, dtype: float64
```

时区转换

```
ts_utc = ts.tz_localize('UTC')
print(ts_utc)
print(ts_utc.tz_convert('US/Eastern'))
```

```
2012-03-06 00:00:00+00:00    0.184221
2012-03-07 00:00:00+00:00    0.020676
2012-03-08 00:00:00+00:00   -0.207159
2012-03-09 00:00:00+00:00   -0.938459
2012-03-10 00:00:00+00:00   -0.723868
Freq: D, dtype: float64
2012-03-05 19:00:00-05:00    0.184221
2012-03-06 19:00:00-05:00    0.020676
2012-03-07 19:00:00-05:00   -0.207159
2012-03-08 19:00:00-05:00   -0.938459
2012-03-09 19:00:00-05:00   -0.723868
Freq: D, dtype: float64
```

2.3.2、分类

将原始等级转换为分类数据类型。

```
df = pd.DataFrame({"id":[1,2,3,4,5,6],
                   "raw_grade":['a', 'b', 'b', 'a', 'a', 'e']})
print(df)
df["grade"] = df["raw_grade"].astype("category")
print(df["grade"])
```

```
   id raw_grade
0   1         a
1   2         b
2   3         b
3   4         a
4   5         a
5   6         e
0    a
1    b
2    b
3    a
4    a
5    e
Name: grade, dtype: category
Categories (3, object): [a, b, e]
```

可以将类别重命名为更有意义的名称

```
df = pd.DataFrame({"id":[1,2,3,4,5,6],
                   "raw_grade":['a', 'b', 'c', 'a', 'a', 'e']})
print(df)
df["grade"] = df["raw_grade"].astype("category")
print(df["grade"])

df["grade"].cat.categories = ["very good", "good","medium", "very bad"]
print(df["grade"])
```

```
0    very good
1         good
2       medium
3    very good
4    very good
5     very bad
Name: grade, dtype: category
Categories (4, object): [very good, good, medium, very bad]
```

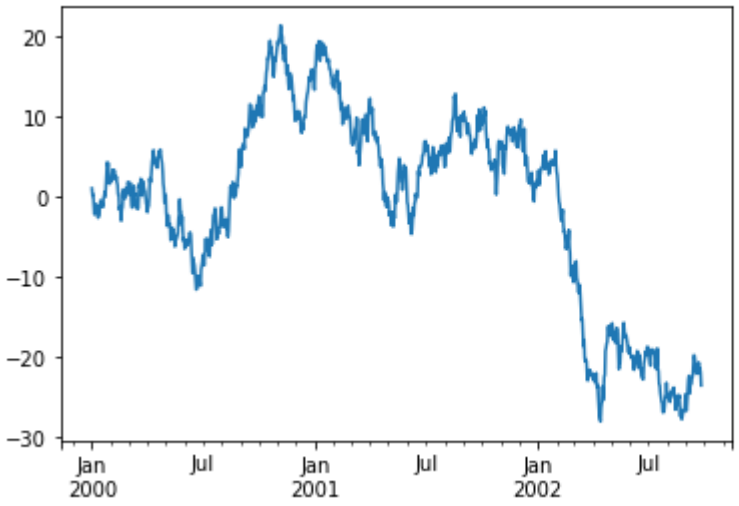
group 操作

```
In [113]: frame1
Out[113]:
     A      B         C
0  foo    one -0.541818
1  bar    one  0.262376
2  foo    two -0.217415
3  bar  three  0.104474
4  foo    two  0.916483
5  bar    two -0.644991
6  foo    two -0.249777
7  bar    one -0.480751

In [114]: frame1.groupby('A')
Out[114]: <pandas.core.groupby.DataFrameGroupBy object at 0x0000028B3449B898>

In [115]: frame1.groupby('A').sum()
Out[115]:
            C
A
bar -0.758891
foo -0.092527
```

```
In [116]: frame1.groupby(['A','B']).sum()
Out[116]:
                  C
A   B
bar one   -0.218374
    three  0.104474
    two   -0.644991
foo one   -0.541818
    two    0.449291
```

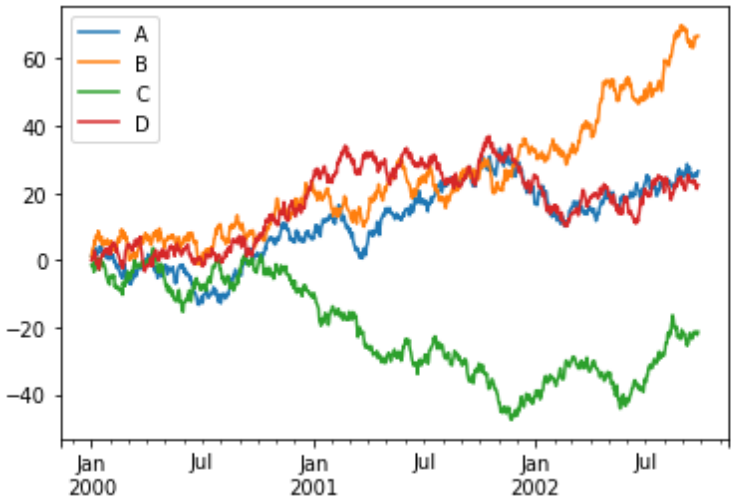
2.3.3、绘图

```
213 ts = pd.Series(np.random.randn(1000),
214                index=pd.date_range('1/1/2000', periods=1000))
215 ts = ts.cumsum()
216 ts.plot()
```



在 DataFrame 上，该 plot()方法可以方便地绘制所有带有标签的列：

```
221 df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
222                   columns=['A', 'B', 'C', 'D'])
223 df = df.cumsum()
224 plt.figure()
225 df.plot()
226 plt.legend(loc='best')
```



2.3.4、数据的输入输出

CSV

```
228 df = pd.DataFrame(np.random.randn(5,2),columns=['A', 'B'])
229 df.to_csv('foo.csv')
230 data = pd.read_csv('foo.csv')
231 print(data)
```

```
   Unnamed: 0         A         B
0           0 -0.934387  0.608668
1           1  0.108335  2.307438
2           2 -0.467820 -0.317032
3           3 -0.965730 -0.558848
4           4  1.135391  0.462674
```

HDF5

```
235 df = pd.DataFrame(np.random.randn(5,2),columns=['A', 'B'])
236 df.to_hdf('foo.h5','df')
237 data = pd.read_hdf('foo.h5','df')
238 print(data)
```

```
          A         B
0 -0.390691 -1.404607
1  0.730216  1.741349
2  1.506964  0.607923
3 -1.021359 -0.734430
4 -0.435500 -0.671622
```

EXCEL

```
242 df = pd.DataFrame(np.random.randn(5,2),columns=['A', 'B'])
243 df.to_excel('foo.xlsx', sheet_name='Sheet1')
244 data = pd.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_values=['NA'])
245 print(data)
```

```
          A         B
0  0.443514  0.134110
1 -0.239742  0.335882
2  1.361807 -0.215797
3 -0.114971  0.058282
4 -0.076920 -1.363760
```

2、sklearn 学习

2.1、sklearn 五个分类算法的调用

2.1.1、逻辑回归

```
import numpy as np
import urllib.request
from sklearn import preprocessing
from sklearn import metrics
from sklearn.linear_model import LogisticRegression


# url with dataset
url = "http://archive.ics.uci.edu/ml/machine-learning-databases/cmc/cmc.data"
# download the file
raw_data = urllib.request.urlopen(url)
# load the CSV file as a numpy matrix
dataset = np.loadtxt(raw_data, delimiter=',')
# separate the data from the target attributes
X = dataset[:, 1:-1]
y = dataset[:, -1]
# normalize the data attributes
normalized_X = preprocessing.normalize(X)
model = LogisticRegression()
```

```
model.fit(normalized_X, y)
print(model)
# make predictions
expected = y
predicted = model.predict(normalized_X)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

结果：

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
             precision    recall  f1-score   support

        1.0       0.54      0.77      0.64       629
        2.0       0.50      0.33      0.39       333
        3.0       0.44      0.31      0.36       511

avg / total       0.50      0.51      0.49      1473

[[485  40 104]
 [125 109  99]
 [281  71 159]]
```

2.1.2、朴素贝叶斯

```
21  model = GaussianNB()
22  model.fit(X, y)
23  print(model)
24  # make predictions
25  expected = y
26  predicted = model.predict(X)
27  # summarize the fit of the model
28  print(metrics.classification_report(expected, predicted))
29  print(metrics.confusion_matrix(expected, predicted))
```

结果：

```
GaussianNB(priors=None)
             precision    recall  f1-score   support

        1.0       0.63      0.43      0.51       629
        2.0       0.36      0.64      0.46       333
        3.0       0.46      0.41      0.43       511

avg / total       0.51      0.47      0.47      1473

[[271 190 168]
 [ 44 214  75]
 [116 188 207]]
```

2.1.3、K 邻近

```
# fit a k-nearest neighbor model to the data
model = KNeighborsClassifier()
model.fit(X, y)
print(model)
# make predictions
expected = y
predicted = model.predict(X)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

结果：

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=5, p=2,
           weights='uniform')
             precision    recall  f1-score   support

        1.0       0.64      0.79      0.71       629
        2.0       0.54      0.53      0.53       333
        3.0       0.70      0.51      0.59       511

avg / total       0.64      0.63      0.63      1473

[[495  58  76]
 [120 176  37]
 [158  92 261]]
```

2.1.4、决策树

```
# fit a CART model to the data
model = DecisionTreeClassifier()
model.fit(X, y)
print(model)
# make predictions
expected = y
predicted = model.predict(X)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
             precision    recall  f1-score   support

        1.0       0.79      0.89      0.84       629
        2.0       0.71      0.72      0.72       333
        3.0       0.83      0.70      0.76       511

avg / total       0.79      0.79      0.78      1473

[[558  33  38]
 [ 58 240  35]
 [ 87  64 360]]
```

2.1.5、支持向量机

```
23      # fit a SVM model to the data
24      model = SVC()
25      model.fit(X, y)
26      print(model)
27      # make predictions
28      expected = y
29      predicted = model.predict(X)
30      # summarize the fit of the model
31      print(metrics.classification_report(expected, predicted))
32      print(metrics.confusion_matrix(expected, predicted))
```

结果：

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
             precision    recall  f1-score   support

        1.0       0.62      0.67      0.65       629
        2.0       0.49      0.43      0.46       333
        3.0       0.52      0.51      0.51       511

avg / total       0.56      0.56      0.56      1473

[[423  61 145]
 [ 96 143  94]
 [163  89 259]]
```

2.2、参数设置

逻辑回归

```
# penalty(正则化选择参数)、solver（优化算法选择参数）、
# multi_class(分类方式选择参数)、class_weight(类型权重)
model = LogisticRegression(penalty='l1',
                           solver='liblinear', multi_class='ovr', class_weight={})
```

朴素贝叶斯

```
# 设置prior先验概率
# model = GaussianNB(priors=...)
```

K 邻近

```
# 设置K值
# model = KNeighborsClassifier(n_neighbors=4)
```

决策树

```
# 设置参数criterion(最优划分属性衡量标准)、max_depth(树的最大深度)
# model = DecisionTreeClassifier(criterion="entropy", max_depth=20)
```

支持向量机

```
# 设置参数kernal、degree、gama
# model = SVC(kernal='rbf', degree=3, gama='auto')
```

2.3、参数优化

2.3.1、网格搜索交叉验证 GridSearchCV

```
# prepare a range of alpha values to test
alphas = np.array([1,0.1,0.01,0.001,0.0001,0])
# create and fit a ridge regression model, testing each alpha
model = Ridge()
grid = GridSearchCV(estimator=model, param_grid=dict(alpha=alphas))
grid.fit(X, y)
print(grid)
# summarize the results of the grid search
print(grid.best_score_)
print(grid.best_estimator_.alpha)
```

结果：

```
GridSearchCV(cv=None, error_score='raise',
       estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001),
       fit_params={}, iid=True, n_jobs=1,
       param_grid={'alpha': array([1.e+00, 1.e-01, 1.e-02, 1.e-03, 1.e-04, 0.e+00])},
       pre_dispatch='2*n_jobs', refit=True, scoring=None, verbose=0)
-5.124095552745394
0.0
```

2.3.2、随机采样交叉验证 RandomizedSearchCV

```
# prepare a uniform distribution to sample for the alpha parameter
param_grid = {'alpha': sp_rand()}
# create and fit a ridge regression model, testing random alpha values
model = Ridge()
rsearch = RandomizedSearchCV(estimator=model, param_distributions=param_grid, n_iter=100)
rsearch.fit(X, y)
print(rsearch)
# summarize the results of the random parameter search
print(rsearch.best_score_)
print(rsearch.best_estimator_.alpha)
```

结果：

```
RandomizedSearchCV(cv=None, error_score='raise',
          estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001),
          fit_params={}, iid=True, n_iter=100, n_jobs=1,
          param_distributions={'alpha': <scipy.stats._distn_infrastructure.rv_frozen object at 0x00000254FFFA1EF0>},
          pre_dispatch='2*n_jobs', random_state=None, refit=True,
          scoring=None, verbose=0)
-5.1240969264002985
0.011690653492592462
```

2.4、交叉验证评估

```
21      from sklearn.model_selection import _validation
22      from sklearn.tree import DecisionTreeClassifier
23      model = DecisionTreeClassifier()
24      score = _validation.cross_val_score(estimator=model, X=X, y=y, cv=10)
25      print(score)
```

结果：

```
 [0.53691275 0.50675676 0.4527027  0.51020408 0.47619048 0.44217687
  0.4829932  0.46258503 0.42857143 0.45205479]
```

得到的 score 位准确率

2.5、sklearn 单机特征工程（包含特征选择）

2.5.1、特征预处理

使用 preprocessing 库解决以下问题

1、不属于同一量纲：无量纲化
2、信息冗余：二值化
3、定性特征不能直接使用：哑编码
4、存在缺失值
5、信息利用率低

```python
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import Normalizer
from sklearn.preprocessing import Binarizer
from sklearn.preprocessing import OneHotEncoder
from numpy import vstack, array, nan
from sklearn.preprocessing import Imputer
from sklearn.preprocessing import PolynomialFeatures
from numpy import log1p
from sklearn.preprocessing import FunctionTransformer


# 导入IRIS数据集
iris = load_iris()
# 特征矩阵
iris.data
# 目标向量
iris.target
```

```python
# 标准化，返回值为标准化后的数据
StandardScaler().fit_transform(iris.data)

# 区间缩放，返回值为缩放到[0, 1]区间的数据
MinMaxScaler().fit_transform(iris.data)

# 归一化，返回值为归一化后的数据
Normalizer().fit_transform(iris.data)

# 二值化，阈值设置为3，返回值为二值化后的数据,简化数据
Binarizer(threshold=3).fit_transform(iris.data)

# 哑编码，对IRIS数据集的目标值，返回值为哑编码后的数据
OneHotEncoder().fit_transform(iris.target.reshape((-1, 1)))

# 由于iris数据集没有缺失值，所以对数据集新增一个样本
# 缺失值计算，返回值为计算缺失值后的数据
# 参数missing_value为缺失值的表示形式，默认为NaN
# 参数strategy为缺失值填充方式，默认为mean（均值）
```

```
Imputer().fit_transform(vstack((array([nan, nan, nan, nan]), iris.data)))

# 多项式转换
# 参数degree为度，默认值为2
PolynomialFeatures().fit_transform(iris.data)

# 自定义转换函数为对数函数的数据变换
# 第一个参数是单变元函数
FunctionTransformer(log1p).fit_transform(iris.data)
```

2.5.2、特征选择

当数据预处理完成后，我们需要选择有意义的特征输入机器学习的算法和模型进行训练。通常来说，从两个方面考虑来选择特征：

1．特征是否发散：如果一个特征不发散，例如方差接近于 0，也就是说样本在这个特征上基本上没有差异，这个特征对于样本的区分并没有什么用。

2．特征与目标的相关性：这点比较显见，与目标相关性高的特征，应当优选选择。除方差法外

根据特征选择的形式又可以将特征选择方法分为 3 种：

1.Filter:过滤法

2.Wrapper:包装法

3.Embedded:嵌入法

使用 sklearn 中的 feature_selection 库来进行特征选择。

```
from array import array
from sklearn.feature_selection import chi2
from sklearn.datasets import load_iris
from sklearn.feature_selection import VarianceThreshold
from scipy.stats import pearsonr
from sklearn.feature_selection import SelectKBest
from minepy import MINE
from sklearn.feature_selection import RFE
from sklearn.feature_selection import SelectFromModel
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import GradientBoostingClassifier

# 导入IRIS数据集
iris = load_iris()

# Filter 按照发散性或者相关性对各个特征进行评分，设定阈值或者待选择阈值的个数，选择特征。
# 方差选择法，返回值为特征选择后的数据
# 参数threshold为方差的阈值
VarianceThreshold(threshold=3).fit_transform(iris.data)

```

```
# 相关系数法，计算各个特征对目标值的相关系数以及相关系数的P值
# 选择K个最好的特征，返回选择特征后的数据
# 第一个参数为计算评估特征是否好的函数，该函数输入特征矩阵和目标向量，
# 输出二元组（评分，P值）的数组，数组第i项为第i个特征的评分和P值。在此定义为计算相关系数
# 参数k为选择的特征个数
SelectKBest(lambda X, Y: array(map(lambda x:pearsonr(x, Y), X.T)).T,
            k=2).fit_transform(iris.data, iris.target)

# 卡方检验，卡方检验是检验定性自变量对定性因变量的相关性。
# 选择K个最好的特征，返回选择特征后的数据
SelectKBest(chi2, k=2).fit_transform(iris.data, iris.target)

# 互信息法用于评价定性自变量对定性因变量的相关性
# 由于MINE的设计不是函数式的，定义mic方法将其为函数式的，返回一个二元组，
# 二元组的第2项设置成固定的P值0.5


def mic(x, y):
    m = MINE()
    m.compute_score(x, y)
    return m.mic(), 0.5


# 选择K个最好的特征，返回特征选择后的数据
SelectKBest(lambda X, Y: array(map(lambda x:mic(x, Y), X.T)).T,
            k=2).fit_transform(iris.data, iris.target)

# Wrapper 根据目标函数（通常是预测效果评分），每次选择若干特征，或者排除若干特征。

# 递归消除特征法使用一个基模型来进行多轮训练，每轮训练后，消除若干权值系数的特征，
# 再基于新的特征集进行下一轮训练。

# 递归特征消除法，返回特征选择后的数据
# 参数estimator为基模型
# 参数n_features_to_select为选择的特征个数
RFE(estimator=LogisticRegression(), n_features_to_select=2).fit_transform(iris.data, iris.target)

# Embedded 嵌入法，先使用某些机器学习的算法和模型进行训练，得到各个特征的权值系数，
# 根据系数从大到小选择特征。类似于Filter方法，但是是通过训练来确定特征的优劣。

# 使用带惩罚项的基模型，除了筛选出特征外，同时也进行了降维
```

```
# 使用feature_selection库的SelectFromModel类结合带L1惩罚项的逻辑回归模型
# 带L1惩罚项的逻辑回归作为基模型的特征选择
SelectFromModel(LogisticRegression(penalty="l1", C=0.1)).fit_transform(iris.data, iris.target)

# 使用feature_selection库的SelectFromModel类结合带L1以及L2惩罚项的逻辑回归模型

# 带L1和L2惩罚项的逻辑回归作为基模型的特征选择
# 参数threshold为权值系数之差的阈值
SelectFromModel(LR(threshold=0.5, C=0.1)).fit_transform(iris.data, iris.target)

# 树模型中GBDT也可用来作为基模型进行特征选择，使用feature_selection库的
# SelectFromModel类结合GBDT模型，来选择特征
# GBDT作为基模型的特征选择
SelectFromModel(GradientBoostingClassifier()).fit_transform(iris.data, iris.target)
```

2.5.3、降维

主要有两种方法，一个是主成分分析法（PCA），一个是线性判别分析法（LDA）

```
from sklearn.decomposition import PCA
# 主成分分析法，返回降维后的数据
# 参数n_components为主成分数目
PCA(n_components=2).fit_transform(iris.data)

from sklearn.lda import LDA
# 线性判别分析法，返回降维后的数据
# 参数n_components为降维后的维数
LDA(n_components=2).fit_transform(iris.data, iris.target)
```

2.6、sklearn 聚类算法的调用

使用 kmeans

```
from sklearn.cluster import KMeans


def create_test_data():
    lines_set = open('E:\myCodes\python\mlTest\\venv\src\data\seeds_dataset.txt').readlines()
    data_x = []
    for data in lines_set:
        temp = [i for i in data.split("\n")[0].split("\t") if i is not '']
        data_x.append(list(map(float, temp)))
    return data_x


clf = KMeans(n_clusters=3)
s = clf.fit(create_test_data())
print(clf.cluster_centers_)
print(clf.labels_)
```

结果：

```
[[11.90906667 13.25026667  0.85154933  5.22233333  2.86509333  4.72218667
   5.09304     2.86666667]
 [14.63202703 14.45324324  0.8790973   5.56178378  3.27489189  2.74404324
   5.18493243  1.13513514]
 [18.72180328 16.29737705  0.88508689  6.20893443  3.72267213  3.60359016
   6.06609836  1.98360656]]
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 2 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 1 2 1 2 2 2 2 2 2 2 1 1 1 1 2 1 1 1 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

3、数据集测试

使用数据集

数据集一：zoo 数据集
数据集二：iris 数据集
数据集三：wine 数据集
数据集四：data_banknote_authentication 数据集
数据集五：transfusion 数据集
数据集六：vowel-context 数据集
数据集七：ecoli 数据集
数据集八：ionosphere 数据集
数据处理

```
# coding=utf-8
from sklearn.model_selection import train_test_split
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
import urllib
from sklearn import metrics
import time
import pandas as pd
from sklearn.linear_model import LogisticRegression
# 通过url获取数据，并获得相应参数
# 返回数据集字典{dataset_name: {data:[],labelColumn:column,source=""}}


def load_data():
    format_data = {}
    url = "http://archive.ics.uci.edu/ml/machine-learning-databases/zoo/zoo.data"
    # 下载文件
    raw_data = urllib.request.urlopen(url)
    # 格式化数据
```

```
    dataset = np.loadtxt(raw_data, delimiter=',', usecols=range(1, 18, 1))
    format_data['zoo数据集'] = {'data': dataset, 'labelColumn': 16, 'multiClass': True}

    url = "http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
    raw_data = urllib.request.urlopen(url)
    df = pd.read_csv(raw_data, delimiter=',')
    dataset = np.array(df.replace(['Iris-setosa', 'Iris-virginica', 'Iris-versicolor'], [0, 1, 2]))
    format_data['iris数据集'] = {'data': dataset, 'labelColumn': 4, 'multiClass': True}

    url = "http://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data"
    raw_data = urllib.request.urlopen(url)
    dataset = np.loadtxt(raw_data, delimiter=',')
    format_data['wine数据集'] = {'data': dataset, 'labelColumn': 0, 'multiClass': True}

    url = "http://archive.ics.uci.edu/ml/machine-learning-databases/00267/da" \
          "ta_banknote_authentication.txt"
    raw_data = urllib.request.urlopen(url)
    dataset = np.loadtxt(raw_data, delimiter=',')
    format_data['data_banknote_authentication数据集'] = {'data': dataset, 'labelColumn': 4
        , 'multiClass': False}
```

```
    url = "http://archive.ics.uci.edu/ml/machine-learning-databases/blood-tr" \
          "ansfusion/transfusion.data"
    raw_data = urllib.request.urlopen(url)
    df = pd.read_csv(raw_data, delimiter=',', skiprows=1)
    format_data['transfusion数据集'] = {'data': np.array(df), 'labelColumn': 4, 'multiClass': False}

    url = "http://archive.ics.uci.edu/ml/machine-learning-databases/undocumented/connec" \
          "tionist-bench/vowel/vowel-context.data"
    raw_data = urllib.request.urlopen(url)
    dataset = np.loadtxt(raw_data)
    format_data['vowel-context数据集'] = {'data': dataset, 'labelColumn': 13, 'multiClass': True}

    url = "http://archive.ics.uci.edu/ml/machine-learning-databases/ecoli/ecoli.data"
    raw_data = urllib.request.urlopen(url)
    df = pd.read_csv(raw_data, delimiter='\s+', usecols=range(1, 9, 1))
    dataset = np.array(df.replace(['cp', 'im', 'pp', 'imU', 'om', 'omL', 'imL', 'imS'], range(0, 8, 1)))
    format_data['ecoli数据集'] = {'data': dataset, 'labelColumn': 7, 'multiClass': True}

    url = "http://archive.ics.uci.edu/ml/machine-learning-databases/ionosphere/ionosphere.data"
    raw_data = urllib.request.urlopen(url)
    dataset = np.array(pd.read_csv(raw_data, delimiter=',').replace(['g', 'b'], [0, 1]))
```

```
    format_data['ionosphere数据集'] = {'data': dataset, 'labelColumn': 34, 'multiClass': False}
    return format_data

```

```
# 使用内置函数划分数据集
def deal_data(data, labelColum):
    y = data[:, labelColum]
    x = np.delete(data, labelColum, axis=1)
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.1)
    return train_x, train_y, test_x, test_y
```

五个分类算法调用

```
77  def logistic_regression_clf(tr_x, tr_y):
78      model = LogisticRegression()
79      model.fit(tr_x, tr_y)
80      return model
81
82
83  def tree_clf(tr_x, tr_y):
84      model = DecisionTreeClassifier()
85      model.fit(tr_x, tr_y)
86      return model
87
88
89  def knn_clf(tr_x, tr_y):
90      model = KNeighborsClassifier()
91      model.fit(tr_x, tr_y)
92      return model
93
94
95  def naive_bayes_clf(tr_x, tr_y):
96      model = GaussianNB()
97      model.fit(tr_x, tr_y)
```

```
98      return model
99
100
101 def svm_clf(tr_x, tr_y):
102     model = SVC()
103     model.fit(tr_x, tr_y)
104     return model
```

开始训练

```
107 if __name__ == '__main__':
108     clf_dic = {
109         '逻辑回归': logistic_regression_clf,
110         '决策树': tree_clf,
111         'k邻近': knn_clf,
112         '朴素贝叶斯': naive_bayes_clf,
113         '支持向量机': svm_clf
114     }
115     data = load_data()
```

```
    for key in data:
        results = []
        train_x, train_y, test_x, test_y = deal_data(data[key]['data'], data[key]['labelColumn'])
        for key1 in clf_dic:
            result = [key1]
            start_time = time.time()
            model = clf_dic[key1](train_x, train_y)
            expected = test_y
            predicted = model.predict(test_x)
            ave = 'binary'
            if data[key]['multiClass']:
                ave = 'macro'
            acc = metrics.accuracy_score(expected, predicted)
            pre = metrics.precision_score(expected, predicted, average=ave)
            recall = metrics.recall_score(expected, predicted, average=ave)
            f1 = metrics.f1_score(expected, predicted, average=ave)
            use_time = time.time() - start_time
            result.append(use_time)
            result.append(pre)
            result.append(recall)
```

```
            result.append(acc)
            result.append(f1)
            results.append(result)
        print('==============', key, '==================')
        df = pd.DataFrame(results, columns=['分类算法', 'time', 'precision',
                                            'recall', 'acc', 'f1_score'])
        print(df)
```

结果

```
============== zoo数据集 ==================
    分类算法      time  precision    recall       acc  f1_score
0   逻辑回归  0.009999        1.0  1.000000  1.000000  1.000000
1    决策树  0.000000        1.0  1.000000  1.000000  1.000000
2    k邻近  0.010000        0.4  0.600000  0.818182  0.466667
3  朴素贝叶斯  0.000000        0.9  0.971429  0.909091  0.917949
4  支持向量机  0.009998        0.7  0.800000  0.909091  0.733333
============== iris数据集 ==================
    分类算法      time  precision    recall       acc  f1_score
0   逻辑回归  0.000000   0.952381  0.944444  0.933333  0.944056
1    决策树  0.000000   0.952381  0.944444  0.933333  0.944056
2    k邻近  0.010001   0.952381  0.944444  0.933333  0.944056
3  朴素贝叶斯  0.000000   0.952381  0.944444  0.933333  0.944056
4  支持向量机  0.000000   0.952381  0.944444  0.933333  0.944056
```

```
============== wine数据集 ==================
    分类算法      time  precision    recall       acc  f1_score
0   逻辑回归  0.000000   0.962963  0.944444  0.944444  0.950089
1    决策树  0.000000   0.952381  0.958333  0.944444  0.952137
2    k邻近  0.009998   0.759259  0.736111  0.777778  0.742764
3  朴素贝叶斯  0.000000   0.962963  0.944444  0.944444  0.950089
4  支持向量机  0.000000   0.490196  0.388889  0.500000  0.308571
============== data_banknote_authentication数据集 ==================
    分类算法      time  precision    recall       acc  f1_score
0   逻辑回归  0.010001   0.982456  1.000000  0.992754  0.991150
1    决策树  0.000000   1.000000  0.982143  0.992754  0.990991
2    k邻近  0.009998   1.000000  1.000000  1.000000  1.000000
3  朴素贝叶斯  0.000000   0.843137  0.767857  0.847826  0.803738
4  支持向量机  0.030001   1.000000  1.000000  1.000000  1.000000
============== transfusion数据集 ==================
    分类算法      time  precision    recall       acc  f1_score
0   逻辑回归  0.000000   0.666667  0.222222  0.786667  0.333333
1    决策树  0.000000   0.470588  0.444444  0.746667  0.457143
2    k邻近  0.010004   0.600000  0.333333  0.786667  0.428571
3  朴素贝叶斯  0.000000   0.363636  0.222222  0.720000  0.275862
4  支持向量机  0.029999   0.333333  0.111111  0.733333  0.166667
```

```
============== vowel-context数据集 ==================
    分类算法      time  precision    recall       acc  f1_score
0   逻辑回归  0.050005   0.561519  0.575620  0.565657  0.547984
1    决策树  0.020001   0.847104  0.847786  0.818182  0.828917
2    k邻近  0.009996   0.944481  0.969658  0.959596  0.952580
3  朴素贝叶斯  0.000000   0.709407  0.704664  0.686869  0.676098
4  支持向量机  0.060001   0.868157  0.879900  0.858586  0.860967
```

```
============== ecoli数据集 ==================
    分类算法      time  precision    recall       acc  f1_score
0   逻辑回归  0.010002   0.515556  0.512121  0.735294  0.507381
1    决策树  0.000000   0.873333  0.896970  0.911765  0.882540
2    k邻近  0.000000   0.575952  0.660606  0.794118  0.600212
3  朴素贝叶斯  0.009998   0.432828  0.449495  0.676471  0.399808
4  支持向量机  0.000000   0.410160  0.495455  0.735294  0.446100
============== ionosphere数据集 ==================
    分类算法      time  precision    recall       acc  f1_score
0   逻辑回归  0.000000   0.900000  0.642857  0.828571  0.750000
1    决策树  0.010000   0.785714  0.785714  0.828571  0.785714
2    k邻近  0.000000   1.000000  0.357143  0.742857  0.526316
3  朴素贝叶斯  0.000000   0.888889  0.571429  0.800000  0.695652
4  支持向量机  0.009999   1.000000  0.785714  0.914286  0.880000
```

4、总结

经过这一阶段的学习，发现自己真的需要一定的知识积累才能游刃有余的操纵数据，从文件数据的读取，到对数据的各种处理，变形，都需要对数据有足够的理解和扎实的编程语言基础。自己对 pandas 和 numpy 的理解还很浅显。需要积累。