使用pandas进行数据预处理

2018/10/5

目录

合并数据

任务描述

- 菜品数据表格数据非常多，如菜品订单详情表、菜品信息表、菜品分类表和用户信息表等，这些数据分别存储了菜品数据分析过程中所需要的各种信息。
- 通过堆叠合并和逐渐合并等多种合并方式，可以将关联的数据信息合并在一张表中。

任务分析

（1）横向或纵向堆叠合并数据

（2）主键合并数据

（3）重叠合并数据

堆叠合并数据

1. 横向表堆叠

- 横向堆叠，即将两个表在X轴向拼接在一起，可以使用**concat**函数完成，concat函数的基本语法如下。

*pandas.**concat**(objs, axis=0, join='outer', join_axes=None, ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False, copy=True)*

- 常用参数如下所示。

参数名称	说明
objs	接收多个Series，DataFrame，Panel的组合。表示参与链接的pandas对象的列表的组合。无默认。
axis	接收0或1。表示连接的轴向，默认为0。
join	接收inner或outer。表示其他轴向上的索引是按交集（inner）还是并集（outer）进行合并。默认为outer。

堆叠合并数据

1. 横向表堆叠

续上表

参数名称	说明
join_axes	接收Index对象。表示用于其他n-1条轴的索引，不执行并集 / 交集运算。
ignore_index	接收boolean。表示是否不保留连接轴上的索引，产生一组新索引range(total_length)。默认为False。
keys	接收sequence。表示与连接对象有关的值，用于形成连接轴向上的层次化索引。默认为None。
levels	接收包含多个sequence的list。表示在指定keys参数后，指定用作层次化索引各级别上的索引。默认为None。
names	接收list。表示在设置了keys和levels参数后，用于创建分层级别的名称。默认为None。
verify_integrity	接收boolearn。表示是否检查结果对象新轴上的重复情况，如果发现则引发异常。默认为False。

堆叠合并数据

1. 横向表堆叠

- 当axis=1的时候，concat做行对齐，然后将不同列名称的两张或多张表合并。当两个表索引不完全一样时，可以使用join参数选择是内连接还是外连接。在内连接的情况下，仅仅返回索引重叠部分。在外连接的情况下，则显示索引的并集部分数据，不足的地方则使用空值填补。

表1

	A	B	C	D
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4

表2

	B	D	F
2	B2	D2	F2
4	B4	D4	F4
6	B6	D6	F6
8	B8	D8	F8

合并后表3

	A	B	C	D	B	D	F
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	NaN	NaN	NaN
4	A4	B4	C4	D4	B4	D4	F4
6	NaN	NaN	NaN	NaN	B6	D6	F6
8	NaN	NaN	NaN	NaN	B8	D8	F8

- 当两张表完全一样时，不论join参数取值是inner或者outer，结果都是将两个表完全按照X轴拼接起来。

堆叠合并数据

1. 横向表堆叠

```
import numpy as np
import pandas as pd
from sqlalchemy import create_engine
conn = create_engine('mysql+pymysql://root:1234@\127.0.0.1:3306/testdb?charset=utf8')
detail1 = pd.read_sql('meal_order_detail1',conn)
df1 = detail1.iloc[:,:10]              ##取出detail1的前10列数据
df2 = detail1.iloc[:,10:]              ##取出detail1的后9列数据
print('合并df1的大小为%s，df2的大小为%s。'%(df1.shape,df2.shape))
print('外连接合并后的数据框大小为：',pd.concat([df1,df2], axis=1,join='inner').shape)
print('内连接合并后的数据框大小为：',pd.concat([df1,df2], axis=1,join='outer').shape)
```

堆叠合并数据

1. 横向表堆叠

```
合并df1的大小为(2779, 10)，df2的大小为(2779, 9)。
外连接合并后的数据框大小为:  (2779, 19)
内连接合并后的数据框大小为:  (2779, 19)
```

```
import numpy as np
import pandas as pd
from sqlalchemy import create_engine
conn = create_engine('mysql+pymysql://root:1234@\127.0.0.1:3306/testdb?charset=utf8')
detail1 = pd.read_sql('meal_order_detail1',conn)
df1 = detail1.iloc[:,:10]              ##取出detail1的前10列数据
df2 = detail1.iloc[:,10:]              ##取出detail1的后9列数据
print('合并df1的大小为%s，df2的大小为%s。'%(df1.shape,df2.shape))
print('外连接合并后的数据框大小为：',pd.concat([df1,df2], axis=1,join='inner').shape)
print('内连接合并后的数据框大小为：',pd.concat([df1,df2], axis=1,join='outer').shape)
```

堆叠合并数据

2. 纵向堆叠——concat函数

- 使用concat函数时，在默认情况下，即axis=0时，concat做列对齐，将不同行索引的两张或多张表纵向合并。在两张表的列名并不完全相同的情况下，join参数取值为inner时，返回的仅仅是列名交集所代表的列，取值为outer时，返回的是两者列名的并集所代表的列，其原理示意如图。
- 不论join参数取值是inner或者outer，结果都是将两个表完全按照Y轴拼接起来

表1

	A	B	C	D
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4

表2

	B	D	F
2	B2	D2	F2
4	B4	D4	F4
6	B6	D6	F6
8	B8	D8	F8

合并后表3

	A	B	C	D	F
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
4	A4	B4	C4	D4	NaN
2	NaN	B2	NaN	D2	F2
4	NaN	B4	NaN	D4	F4
6	NaN	B6	NaN	D6	F6
8	NaN	B8	NaN	D8	F8

堆叠合并数据

2. 纵向堆叠——concat函数

```
df3 = detail1.iloc[:1500,:] ##取出detail1前1500行数据
df4 = detail1.iloc[1500:,:] ##取出detail1的1500后的数据
print('合并df3的大小为%s，df4的大小为%s。'%(df3.shape,df4.shape))
print('内连接纵向合并后的数据框大小为：',pd.concat([df3,df4], axis=0,join='inner').shape)
print('外连接纵向合并后的数据框大小为：',pd.concat([df3,df4], axis=0,join='outer').shape)
```

```
合并df3的大小为(1500, 19)，df4的大小为(1279, 19)。
内连接纵向合并后的数据框大小为： (2779, 19)
外连接纵向合并后的数据框大小为： (2779, 19)
```

堆叠合并数据

2. 纵向堆叠——append方法

- append方法也可以用于纵向合并两张表。但是append方法实现纵向表堆叠有一个前提条件，那就是两张表的列名需要完全一致。append方法的基本语法如下

  *pandas.DataFrame.**append**(self, other, ignore_index=False, verify_integrity=False)*。

- 常用参数如下所示。

参数名称	说明
other	接收DataFrame或Series。表示要添加的新数据。无默认。
ignore_index	接收boolean。如果输入True，会对新生成的DataFrame使用新的索引（自动产生）而忽略原来数据的索引。默认为False。
verify_integrity	接收boolean。如果输入True，那么当ignore_index为False时，会检查添加的数据索引是否冲突，如果冲突，则会添加失败。默认为False。

堆叠合并数据

2. 纵向堆叠——append方法

print('堆叠前df3的大小为%s，df4的大小为%s。'%(df3.shape,df4.shape))

print('append纵向堆叠后的数据框大小为：',df3.append(df4).shape)

```
In [12]: print('堆叠前df3的大小为%s，df4的大小为%s。'%(df3.shape,df4.shape))
    ...: print('append纵向堆叠后的数据框大小为：',df3.append(df4).shape)
堆叠前df3的大小为(1500, 19)，df4的大小为(1279, 19)。
append纵向堆叠后的数据框大小为： (2779, 19)
```

主键合并数据

主键合并

- 主键合并，即通过一个或多个键将两个数据集的行连接起来，类似于SQL中的JOIN。
- 针对两张包含不同字段的表，但存在同一个主键，将其根据某几个字段一一对应拼接起来，结果集的列数为两个元数据的列数和减去连接键的数量。merge函数和join方法

左表1

	A	B	Key
1	A1	B1	k1
2	A2	B2	k2
3	A3	B3	k3
4	A4	B4	k4

右表2

	C	D	Key
1	C1	D1	k1
2	C2	D2	k2
3	C3	D3	k3
4	C4	D4	k4

合并后表3

	A	B	Key	C	D
1	A1	B1	k1	C1	D1
2	A2	B2	k2	C2	D2
3	A3	B3	k3	C3	D3
4	A4	B4	k4	C4	D4

- **merge函数**
- **join方法**

都可以实现主键合并

主键合并数据

主键合并——merge函数

- 和数据库的join一样，merge函数也有左连接（left）、右连接（right）、内连接（inner）和外连接（outer），但比起数据库SQL语言中的join和merge函数还有其自身独到之处，例如可以在合并过程中对数据集中的数据进行排序等。

*pandas.**merge**(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'), copy=True, indicator=False)*

- 可根据merge函数中的参数说明，并按照需求修改相关参数，就可以多种方法实现主键合并。

主键合并数据

常用参数及其说明

参数名称	说明
left	接收DataFrame或Series。表示要添加的新数据。无默认。
right	接收DataFrame或Series。表示要添加的新数据。无默认。。
how	接收inner，outer，left，right。表示数据的连接方式。默认为inner。
on	接收string或sequence。表示两个数据合并的主键（必须一致）。默认为None。
left_on	接收string或sequence。表示left参数接收数据用于合并的主键。默认为None。
right_on	接收string或sequence。表示right参数接收数据用于合并的主键。默认为None。
left_index	接收boolean。表示是否将left参数接收数据的index作为连接主键。默认为False。
right_index	接收boolean。表示是否将right参数接收数据的index作为连接主键。默认为False。
sort	接收boolean。表示是否根据连接键对合并后的数据进行排序。默认为False。
suffixes	接收接收tuple。表示用于追加到left和right参数接收数据重叠列名的尾缀默认为('_x', '_y')。

主键合并数据

主键合并——merge函数

```
order = pd.read_csv('data/meal_order_info.csv ‘,sep=',',encoding='gb18030')  ##读取订单信息表
order['info_id'] = order['info_id'].astype('str')                ##info_id转换为字符串格式，为合并做准备
## 订单详情表和订单信息表都有订单编号
##在订单详情表中为order_id，在订单信息表中为info_id
order_detail = pd.merge(detail1,order, left_on='order_id',right_on = 'info_id')
print('detail1订单详情表的原始形状为：',detail1.shape)
print('order订单信息表的原始形状为：',order.shape)
print('订单详情表和订单信息表主键合并后的形状为：',order_detail.shape)
```

```
detail1订单详情表的原始形状为:  (2779, 19)
order订单信息表的原始形状为:  (945, 21)
订单详情表和订单信息表主键合并后的形状为:  (2779, 40)
```

主键合并数据

主键合并——join方法

- join方法也可以实现部分主键合并的功能，但是join方法使用时，两个主键的名字必须相同。

  pandas.DataFrame.join(self, other, on=None, how='left', lsuffix='', rsuffix='', sort=False)

- 常用参数说明如下。

参数名称	说明
other	接收DataFrame、Series或者包含了多个DataFrame的list。表示参与连接的其他DataFrame。无默认。
on	接收列名或者包含列名的list或tuple。表示用于连接的列名。默认为None。
how	接收特定string。inner代表内连接；outer代表外连接；left和right分别代表左连接和右连接。默认为inner。
lsuffix	接收sring。表示用于追加到左侧重叠列名的末尾。无默认。
rsuffix	接收string。表示用于追加到右侧重叠列名的末尾。无默认。
sort	根据连接键对合并后的数据进行排序，默认为True。

主键合并数据

主键合并——join方法

```
order = pd.read_csv('data/meal_order_info.csv ‘, sep=',',encoding='gb18030') ##读取订单信息表
detail1 = pd.read_sql('meal_order_detail1',conn)


order.rename(columns={'info_id':'order_id'},inplace=True)
##order['order_id'] = pd.to_numeric(order['order_id'], errors='coerce')
detail1['order_id'] = pd.to_numeric(detail1['order_id'], errors='coerce')
order_detail1 = detail1.join(order,on='order_id',rsuffix='1')
print('订单详情表和订单信息表join合并后的形状为：',order_detail1.shape)
```

```
In [68]: print('订单详情表和订单信息表join合并后的形状为: ',order_detail1.shape)
订单详情表和订单信息表join合并后的形状为:  (2779, 40)
```

重叠合并数据

combine_first方法

- 数据分析和处理过程中偶尔会出现两份数据的内容几乎一致的情况，但是某些特征在其中一张表上是完整的，而在另外一张表上的数据则是缺失的。
- 这时可以采用将数据一对一比较，然后进行填充。除此之外，可以用combine_first方法进行重叠数据合并，其原理如下。

表8

	0	1	2
0	NaN	3.0	5.0
1	NaN	4.6	NaN
2	NaN	7.0	NaN

表9

	0	1	2
1	42	NaN	8.2
2	10	7.0	4.0

合并后表10

	0	1	2
0	NaN	3.0	5.0
1	42	4.6	8.2
2	10	7.0	4.0

重叠合并数据

combine_first方法

- 重叠合并在其他工具或者语言中并不常见，但是pandas库的开发者希望pandas能够解决几乎所有的数据分析问题，因此提供了combine_first方法来进行重叠数据合并。
- combine_first的具体用法：*pandas.DataFrame.**combine_first**(other)*
- 参数及其说明如下。

参数名称	说明
other	接收DataFrame。表示参与重叠合并的另一个DataFrame。无默认。

重叠合并数据

combine_first方法

```
##建立两个字典，除了ID外，别的特征互补
dict1 = {'ID':[1,2,3,4,5,6,7,8,9],
         'System':['win10','win10',np.nan,'win10',np.nan,np.nan,'win7','win7','win8'],
         'cpu':['i7','i5',np.nan,'i7',np.nan,np.nan,'i5','i5','i3']}
dict2 = {'ID':[1,2,3,4,5,6,7,8,9],
         'System':[np.nan,np.nan,'win7',np.nan,'win8','win7',np.nan,np.nan,np.nan],
         'cpu':[np.nan,np.nan,'i3',np.nan,'i7  ','i5',np.nan,np.nan,np.nan]}
## 转换两个字典为DataFrame
df5 = pd.DataFrame(dict1)
df6 = pd.DataFrame(dict2)
print('经过重叠合并后的数据为：\n',df5.combine_first(df6))
```

重叠合并数据

combine_first方法

```
##建立两个字典，除了ID外，别的特征互补
dict1 = {'ID':[1,2,3,4,5,6,7,8,9],
         'System':['win10','win10',np.nan,'win10',np.nan,np.nan,'win
         'cpu':['i7','i5',np.nan,'i7',np.nan,np.nan,'i5','i5','i3']}
dict2 = {'ID':[1,2,3,4,5,6,7,8,9],
         'System':[np.nan,np.nan,'win7',np.nan,'win8','win7',np.nan,
         'cpu':[np.nan,np.nan,'i3',np.nan,'i7  ','i5',np.nan,np.nan,np.n
## 转换两个字典为DataFrame
df5 = pd.DataFrame(dict1)
df6 = pd.DataFrame(dict2)
print('经过重叠合并后的数据为：\n',df5.combine_first(df6))
```

```
经过重叠合并后的数据为:
   ID System cpu
0   1  win10  i7
1   2  win10  i5
2   3   win7  i3
3   4  win10  i7
4   5   win8  i7
5   6   win7  i5
6   7   win7  i5
7   8   win7  i5
8   9   win8  i3
```

任务实现

1.堆叠不同时间的订单详情表

- 订单详情表meal_order_detail1、meal_order_detail2、meal_order_detail3具有相同的特征，但数据时间不同，订单编号也不同，在数据分析过程中需要使用全量数据，故需要将几张表做纵向堆叠操作。

2.主键合并订单详情表、订单信息表和客户信息表

- 订单详情表、订单信息表和客户信息表两两之间存在相同意义的字段，因此需通过主键合并的方式将三张表合并为一张宽表。

任务实现

1.堆叠不同时间的订单详情表

- 订单详情表meal_order_detail1、meal_order_detail2、meal_order_detail3具有相同的特征，但数据时间不同，订单编号也不同，在数据分析过程中需要使用全量数据，故需要将几张表做纵向堆叠操作。

```
import numpy as np
import pandas as pd
from sqlalchemy import create_engine
conn = create_engine('mysql+pymysql://root:1234@\127.0.0.1:3306/testdb?charset=utf8')
detail1 = pd.read_sql('meal_order_detail1',conn)          ## 读取数据
detail2 = pd.read_sql('meal_order_detail2',conn)
detail3 = pd.read_sql('meal_order_detail3',conn)
detail = detail1.append(detail2)                          ## 纵向堆叠三张表
detail = detail.append(detail3)
print('三张订单详情表合并后的形状为：', detail.shape)
```

```
三张订单详情表合并后的形状为： (10037, 19)
```

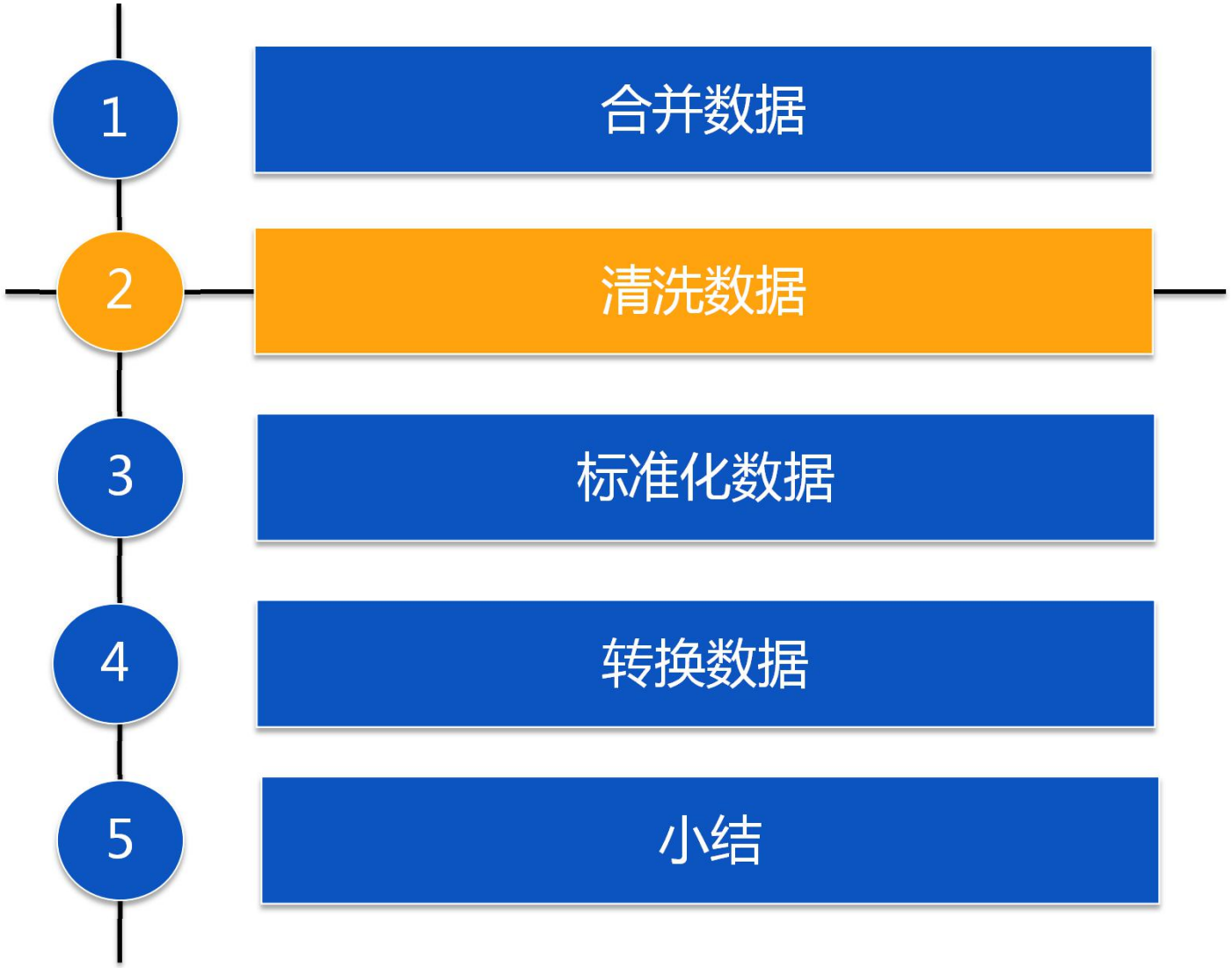
任务实现

2.主键合并订单详情表、订单信息表和客户信息表

- 订单详情表、订单信息表和客户信息表两两之间存在相同意义的字段，因此需通过主键合并的方式将三张表合并为一张宽表。

```
order = pd.read_csv('data/meal_order_info.csv', sep=',',encoding='gb18030')   ##读取订单信息表
user = pd.read_excel('data/users_info.xlsx')                                   ##读取用户信息表
order['info_id'] = order['info_id'].astype('str')                              ## 数据类型转换，存储部分数据
order['emp_id'] = order['emp_id'].astype('str')
user['USER_ID'] = user['USER_ID'].astype('str')
data = pd.merge(detail,order,left_on=['order_id','emp_id'], right_on = ['info_id','emp_id'])
data = pd.merge(data,user,left_on='emp_id ‘, right_on = 'USER_ID',how = 'inner')
print('三张表数据主键合并后的大小为：',data.shape)
```

目录



合并数据

任务描述

- 数据重复会导致数据的方差变小，数据分布发生较大变化。缺失会导致样本信息减少，不仅增加了数据分析的难度，而且会导致数据分析的结果产生偏差。异常值则会产生“伪回归”。因此需要对数据进行检测，查询是否有重复值、缺失值和异常值，并且要对这些数据进行适当的处理。

任务分析

(1) 检测与处理重复值。

(2) 检测与处理数据的缺失值。

(3) 检测与处理数据的异常值

检测与处理重复值

- 处理重复数据是数据分析经常面对的问题之 一。
- 对重复数据进行处理前，需要分析重复数据产生的原因以及去除这部分数据后可能造成的不良影响。
- 常见的数据重复分为两种：
  - 一种为记录重复， 即一个或者多个特征的某几条记录的值完全相同；（行相同）
  - 另一种为特征重复，即存在一个或者多个特征名称不同，但数据完全相同的情况。（列相同）

检测与处理重复值

1.记录重复：即一个或者多个特征某几个记录的值完全相同

- 菜品订单详情表中的 dishes_name 特征存放了每个订单的菜品。要找出所有已点菜品，最简单的方法就是利用去重操作实现。
- 方法一：是利用列表（list）去重，自定义去重函数：

```
def delRep(list1):
    list2=[]
    for i in list1:
        if i not in list2:
            list2.append(i)
    return list2
```

检测与处理重复值

1.记录重复：即一个或者多个特征某几个记录的值完全相同

```
detail = pd.read_csv('data/detail.csv', index_col=0,encoding = 'gbk')
dishes=list(detail['dishes_name'])    ##将dishes_name从数据框中提取出来
print('去重前菜品总数为：',len(dishes))
dish = delRep(dishes)          ##使用自定义的去重函数去重
print('方法一去重后菜品总数为：',len(dish))
```

```
去重前菜品总数为：  10037
方法一去重后菜品总数为：  145
```

```
##自定义去重函数
def delRep(list1):
    list2=[]
    for i in list1:
        if i not in list2:
            list2.append(i)
    return list2
```

检测与处理重复值

1.记录重复：即一个或者多个特征某几个记录的值完全相同

- 方法二是利用集合（set）的元素是唯一的特性去重

```
detail = pd.read_csv('data/detail.csv', index_col=0,encoding = 'gbk')
dishes=list(detail['dishes_name'])    ##将dishes_name从数据框中提取出来
print('去重前菜品总数为：',len(dishes))
dish_set = set(dishes)              ##利用set的特性去重
print('方法二去重后菜品总数为：',len(dish_set))
```

```
去重前菜品总数为:  10037
方法二去重后菜品总数为:  145
```

检测与处理重复值

1.记录重复

- 比较上述两种方法可以发现，方法一代码冗长，会拖慢整个数据分析的进度。
- 方法二使用了集合元素的唯一性，看似代码简单了许多，但会导致数据的排列发生改变。
- 不同方法去前后的部分数据排列比较：

源数据	方法一去重后数据	方法二去重后数据
蒜蓉生蚝	蒜蓉生蚝	纸杯蛋糕
蒙古烤羊腿	蒙古烤羊腿	爆炒鳝碌
大蒜苋菜	大蒜苋菜	黄尾袋鼠西拉子红葡萄酒
芝麻烤紫菜	芝麻烤紫菜	白斩鸡
蒜香包	蒜香包	香菇鸡肉粥
白斩鸡	白斩鸡	农夫山泉NFC果汁100%橙汁

检测与处理重复值

1.记录重复

- pandas提供了一个名为drop_duplicates的去重方法。该方法只对DataFrame或者Series类型有效。
- 这种方法不会改变数据原始排列，并且兼具代码简洁和运行稳定的特点。
- 该方法不仅支持单一特征的数据去重，还能够依据DataFrame的其中一个或者几个特征进行去重操作。

*pandas.DataFrame(Series).**drop_duplicates**(self, subset=None, keep='first', inplace=False)*

参数名称	说明
subset	接收string或sequence。表示进行去重的列。默认为None，表示全部列。
keep	接收特定string。表示重复时保留第几个数据。 First：保留第一个。Last：保留最后一个。False：只要有重复都不保留。默认为first。
inplace	接收boolean。表示是否在原表上进行操作。默认为False。

检测与处理重复值

1.记录重复

- 对菜品订单详情表中的 dishes_name 列利用 drop_duplicates 方法进行去重操作

```
dishes_name = detail['dishes_name'].drop_duplicates()
print('drop_duplicates方法去重之后菜品总数为：',len(dishes_name))
```

```
In [19]: dishes_name = detail['dishes_name'].drop_duplicates()
    ...: print('drop_duplicates方法去重之后菜品总数为: ',len(dishes_name))
drop_duplicates方法去重之后菜品总数为:  145
```

检测与处理重复值

1.记录重复

- 事实上，**drop_duplicates** 方法不仅支持单一特征的数据去重，还能够依据DataFrame中一个或者几个特征进行去重操作，如下：

```
print('去重之前订单详情表的形状为：', detail.shape)
shapeDet = detail.drop_duplicates(subset = ['order_id','emp_id']).shape
print('依照订单编号，会员编号去重之后订单详情表大小为:', shapeDet)
```

```
In [18]: print('去重之前订单详情表的形状为: ', detail.shape)
    ...: shapeDet = detail.drop_duplicates(subset = ['order_id','emp_id']).shape
    ...: print('依照订单编号，会员编号去重之后订单详情表大小为:', shapeDet)
去重之前订单详情表的形状为:  (10037, 18)
依照订单编号，会员编号去重之后订单详情表大小为: (942, 18)
```

检测与处理重复值

2. 特征重复

- 结合相关的数学和统计学知识，去除连续型特征重复可以利用特征间的相似度将两个相似度为1的特征去除一个。（降维）
- 在pandas中相似度的计算方法为 corr，使用该方法计算相似度时，默认为“pearson”法 ；
- 可以通过“method”参数调节成其他方法，目前还支持“spearman”法和“kendall”法。

统计相关系数简介（补充）

相关系数：考察两个事物（在数据里我们称之为变量）之间的相关程度

- 如果有两个变量：X、Y，最终计算出的相关系数的含义可以有如下理解：

(1)、当相关系数为0时，X和Y两变量无关系。

(2)、当X的值增大（减小），Y值增大（减小），两个变量为正相关，相关系数在0.00与1.00之间。

(3)、当X的值增大（减小），Y值减小（增大），两个变量为负相关，相关系数在-1.00与0.00之间。

- 相关系数的绝对值越大，相关性越强，相关系数越接近于1或-1，相关度越强，相关系数越接近于0，相关度越弱。
- 通常情况下通过以下取值范围判断变量的相关强度：

  0.8-1.0 极强相关；　0.6-0.8 强相关　0.4-0.6 中等程度相关

  0.2-0.4 弱相关；　0.0-0.2 极弱相关或无相关

统计相关系数简介（补充）

Pearson（皮尔逊）相关系数

- 皮尔逊相关也称为积差相关（或积矩相关）是英国统计学家皮尔逊于20世纪提出的一种计算直线相关的方法。
- 公式一：$$\rho_{X,Y}=\frac{\operatorname{cov}(X,Y)}{\sigma_X\sigma_Y}=\frac{E((X-\mu_X)(Y-\mu_Y))}{\sigma_X\sigma_Y}=\frac{E(XY)-E(X)E(Y)}{\sqrt{E(X^2)-E^2(X)}\sqrt{E(Y^2)-E^2(Y)}}$$
- 公式二：$$\rho_{X,Y}=\frac{N\sum XY-\sum X\sum Y}{\sqrt{N\sum X^2-(\sum X)^2}\sqrt{N\sum Y^2-(\sum Y)^2}}$$ 公式三：$$\rho_{X,Y}=\frac{\sum(X-\overline{X})(Y-\overline{Y})}{\sqrt{\sum(X-\overline{X})^2\sum(Y-\overline{Y})^2}}$$
- 公式四：$$\rho_{X,Y}=\frac{\sum XY-\frac{\sum X\sum Y}{N}}{\sqrt{(\sum X^2-\frac{(\sum X)^2}{N})(\sum Y^2-\frac{(\sum Y)^2}{N})}}$$
- **以上列出的四个公式等价，其中E是数学期望，cov表示协方差，N表示变量取值的个数。**

统计相关系数简介（补充）

Pearson（皮尔逊）相关系数

- **适用范围：**当两个变量的标准差都不为零时，相关系数才有定义，皮尔逊相关系数适用于：

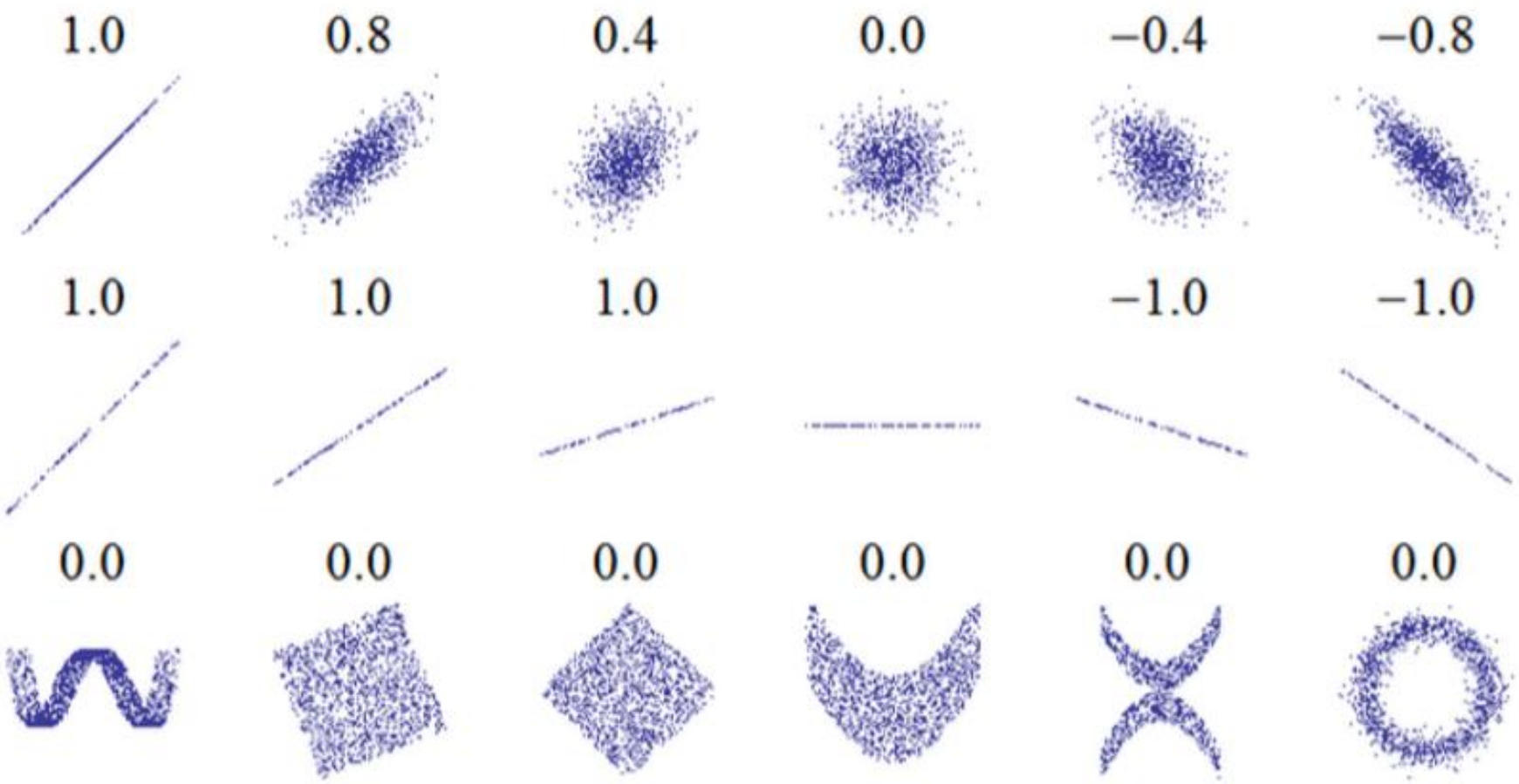
(1)、两个变量之间是线性关系，都是连续数据。

(2)、两个变量的总体是正态分布，或接近正态的单峰分布。

(3)、两个变量的观测值是成对的，每对观测值之间相互独立。

统计相关系数简介（补充）

Pearson（皮尔逊）相关系数



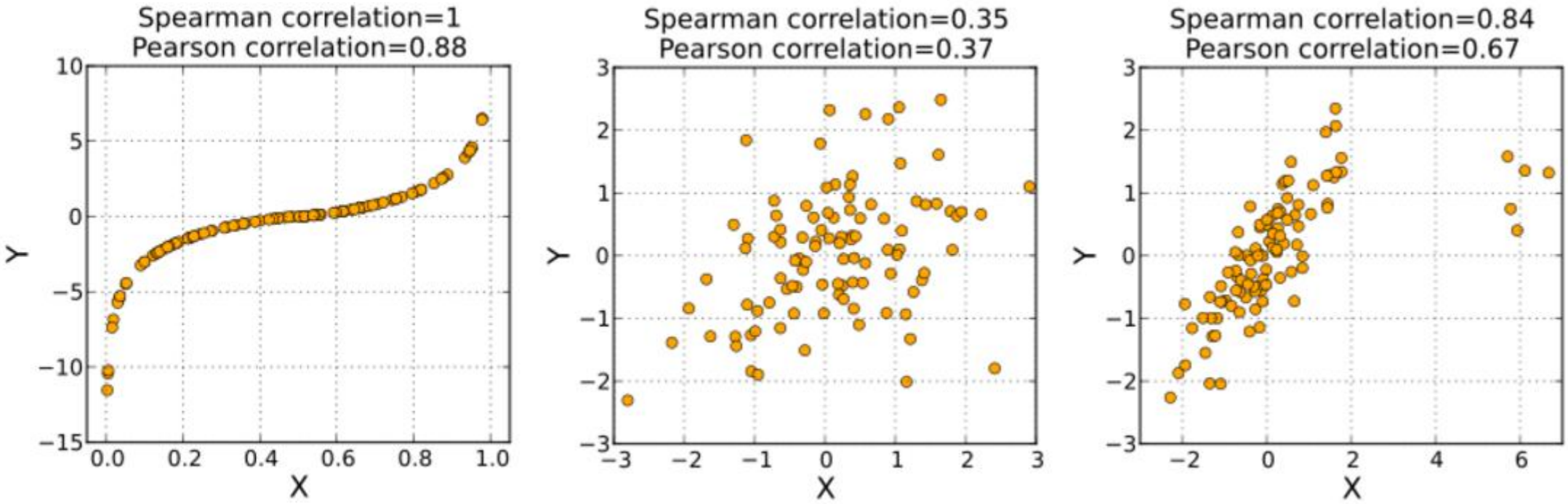
统计相关系数简介（补充）

Spearman Rank（斯皮尔曼等级）相关系数

- 在统计学中，斯皮尔曼等级相关系数以Charles Spearman命名，并经常用希腊字母ρ（rho）表示其值。
- 斯皮尔曼等级相关系数用来估计两个变量X、Y之间的相关性，其中变量间的相关性可以使用单调函数来描述。如果两个变量取值的两个集合中均不存在相同的两个元素，那么，当其中一个变量可以表示为另一个变量的很好的单调函数时（即两个变量的变化趋势相同），两个变量之间的ρ可以达到+1或-1。
- **Spearman（斯皮尔曼）**等级相关系数对数据条件的要求没有**Pearson（皮尔逊）**相关系数严格，只要两个变量的观测值是成对的等级评定资料，或者是由连续变量观测资料转化得到的等级资料，不论两个变量的总体分布形态、样本容量的大小如何，都可以用斯皮尔曼等级相关系数来进行研究

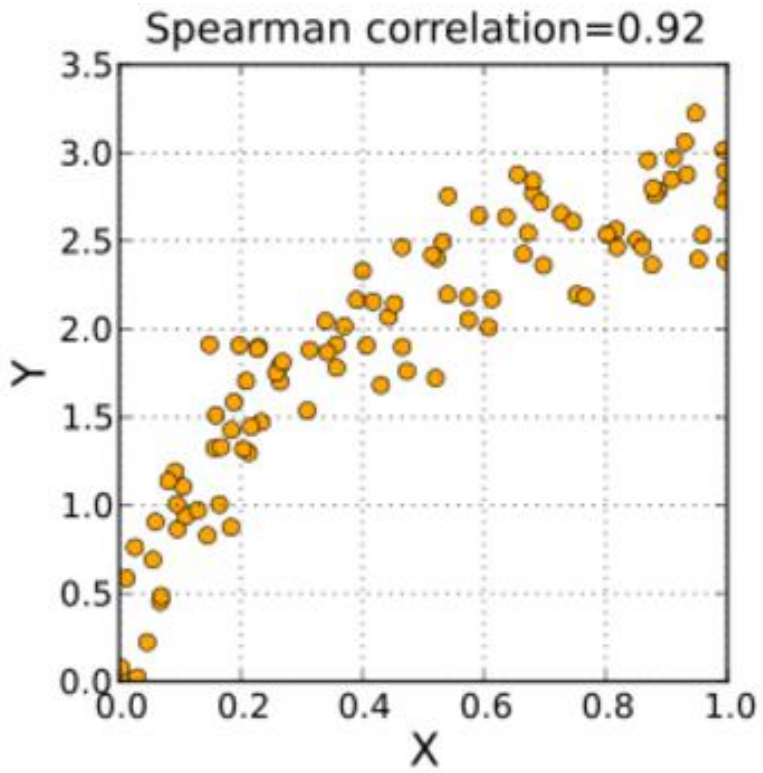
统计相关系数简介（补充）

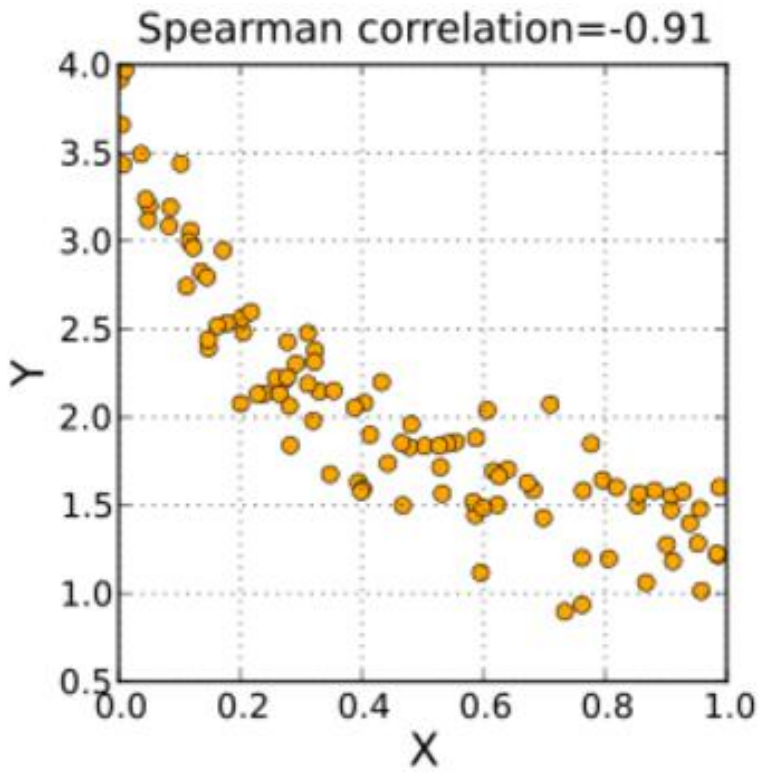
Spearman Rank（斯皮尔曼等级）相关系数



统计相关系数简介（补充）

Spearman Rank（斯皮尔曼等级）相关系数





统计相关系数简介（补充）

Kendall Rank（肯德尔等级）相关系数

- 在统计学中，肯德尔相关系数是以Maurice Kendall命名的，并经常用希腊字母τ（tau）表示其值。肯德尔相关系数是一个用来测量两个随机变量相关性的统计值。一个肯德尔检验是一个无参数假设检验，它使用计算而得的相关系数去检验两个随机变量的统计依赖性。
- 肯德尔相关系数的取值范围在-1到1之间，当τ为1时，表示两个随机变量拥有一致的等级相关性；当τ为-1时，表示两个随机变量拥有完全相反的等级相关性；当τ为0时，表示两个随机变量是相互独立的。

检测与处理重复值

2. 特征重复

- 使用 “kendall”法求出菜品订单详情表（midetail1.xlsx）数据中counts列和amounts列的相似度矩阵。

```
## 求取销量和售价的相似度
corrDet = detail[['counts','amounts']].corr(method='kendall')
print('销量和售价的kendall相似度为：\n',corrDet)
```

```
In [6]: corrDet = detail[['counts','amounts']].corr(method='kendall')
   ...: print('销量和售价的kendall相似度为: \n',corrDet)
销量和售价的kendall相似度为:
             counts    amounts
counts    1.000000 -0.229968
amounts -0.229968  1.000000
```

检测与处理重复值

2. 特征重复

- 但是通过相似度矩阵去重存在一个弊端，该方法只能对数值型重复特征去重，类别型特征之间无法通过计算相似系数来衡量相似度，因此无法根据相似度矩阵对其进行去重处理。
- 对订单详情表中的dishes_name、counts和amounts这3个特征进行pearson法相似度矩阵的求解，但是最终却只能在counts和amounts特征2*2的相似度矩阵。

```
In [7]: corrDet1 = detail[['dishes_name','counts','amounts']].corr(method='pearson')
   ...: print('菜品名称，销量和售价的pearson相似度为: \n',corrDet1)
菜品名称，销量和售价的pearson相似度为:
             counts    amounts
counts    1.000000 -0.159264
amounts -0.159264  1.000000
```

检测与处理重复值

2. 特征重复

- 除了使用相似度矩阵进行特征去重之外，可以通过DataFrame.equals的方法进行特征去重。

```
##定义求取特征是否完全相同的矩阵的函数
def FeatureEquals(df):
    dfEquals=pd.DataFrame([],columns=df.columns,index=df.columns)  ##建立一个DataFrame
    for i in df.columns:
        for j in df.columns:
            dfEquals.loc[i,j]=df.loc[:,i].equals(df.loc[:,j])
    return dfEquals
```

```
## 应用上述函数
detEquals=FeatureEquals(detail)
print('detail的特征相等矩阵的前5行5列为：\n',detEquals.iloc[:5,:5])
```

检测与处理重复值

2. 特征重复

- 除了使用相似度矩阵进行特征去重之外，可以通过 DataFrame.equals 的方法进行特征去重。

```
In [9]: detEquals=FeatureEquals(detail)
   ...: print('detail的特征相等矩阵的前5行5列为: \n',detEquals.iloc[:5,:5])
detail的特征相等矩阵的前5行5列为:
                    order_id     ...       dishes_name
order_id               True      ...             False
dishes_id             False      ...             False
logicprn_name         False      ...             False
parent_class_name     False      ...             False
dishes_name           False      ...              True

[5 rows x 5 columns]
```

检测与处理重复值

2. 特征重复

- 再通过遍历的方式筛选出完全重复的特征

```
##遍历所有数据
lenDet = detEquals.shape[0]
dupCol = []
for k in range(lenDet):
    for l in range(k+1,lenDet):
        if detEquals.iloc[k,l] & (detEquals.columns[l] not in dupCol):
            dupCol.append(detEquals.columns[l])
##进行去重操作
print('需要删除的列为：',dupCol)
detail.drop(dupCol,axis=1,inplace=True)
print('删除多余列后detail的特征数目为：',detail.shape[1])
```

检测与处理重复值

2. 特征重复

- 再通过遍历的方式筛选出完全重复的特征

```
In [12]: lenDet = detEquals.shape[0]
    ...: dupCol = []
    ...: for k in range(lenDet):
    ...:     for l in range(k+1,lenDet):
    ...:         if detEquals.iloc[k,l] & (detEquals.columns[l] not in dupCol):
    ...:             dupCol.append(detEquals.columns[l])
    ...:
    ...: ##进行去重操作
    ...: print('需要删除的列为: ',dupCol)
    ...: detail.drop(dupCol,axis=1,inplace=True)
    ...: print('删除多余列后detail的特征数目为: ',detail.shape[1])
需要删除的列为:  ['parent_class_name', 'cost', 'discount_amt', 'discount_reason',
'kick_back', 'add_info', 'bar_code', 'add_inprice']
删除多余列后detail的特征数目为:  10
```

检测与处理缺失值

Why + How

- 数据中的某个或某些特征的值是不完整的，这些值称为缺失值。
- pandas提供了识别缺失值的方法 **isnull** 以及识别非缺失值的方法 **notnull**，这两种方法在使用时返回的都是布尔值 True 和 False。
- 结合sum函数和 isnull、notnull 函数，可以检测数据中缺失值的分布以及数据中一共含有多少缺失值。
- Isnull 和 notnull 之间结果正好相反，因此使用其中任意一个都可以判断出数据中缺失值的位置。

检测与处理缺失值

利用 isnull 或 notnull 找到缺失值

```
print('detail每个特征缺失的数目为：\n',detail.isnull().sum())
print('detail每个特征非缺失的数目为：\n',detail.notnull().sum())
```

```
In [13]: print('detail每个特征缺失的数目为: \n',detail.isnull().sum())
    ...: print('detail每个特征非缺失的数目为: \n',detail.notnull().sum())
detail每个特征缺失的数目为:
 order_id          0
dishes_id          0
logicprn_name  10037
dishes_name        0
itemis_add         0
counts             0
amounts            0
place_order_time   0
picture_file       0
emp_id             0
dtype: int64
detail每个特征非缺失的数目为:
 order_id          10037
dishes_id          10037
logicprn_name          0
dishes_name        10037
itemis_add         10037
counts             10037
amounts            10037
place_order_time   10037
picture_file       10037
emp_id             10037
dtype: int64
```

检测与处理缺失值

1. 删除法：将含有缺失值的特征或者记录删除

- 删除法分为删除观测记录（行）和删除特征（列）两种，它属于利用减少样本量来换取信息完整度的一种方法，是一种最简单的缺失值处理方法。
- pandas中提供了简便的删除缺失值的方法dropna，该方法既可以删除观测记录(axis=0)，亦可以删除特征(axis=1)。

  *pandas.DataFrame.**dropna**(self, axis=0, how='any', thresh=None, subset=None, inplace=False)*
- 常用参数及其说明如下。

参数名称	说明
axis	接收0或1。表示轴向，0为删除观测记录（行），1为删除特征（列）。默认为0。
how	接收特定string。表示删除的形式。any表示只要有缺失值存在就执行删除操作。all表示当且仅当全部为缺失值时执行删除操作。默认为any。
subset	接收类array数据。表示进行去重的列/行。默认为None，表示所有列/行。
inplace	接收boolean。表示是否在原表上进行操作。默认为False。

检测与处理缺失值

1. 删除法：将含有缺失值的特征或者记录删除

```
print('去除缺失的列前detail的形状为：', detail.shape)
print('去除缺失的列后detail的形状为：  ', detail.dropna(axis = 1,how ='any').shape)
```

```
In [14]: print('去除缺失的列前detail的形状为: ', detail.shape)
    ...: print('去除缺失的列后detail的形状为: ',
    ...:       detail.dropna(axis = 1,how ='any').shape)
去除缺失的列前detail的形状为:  (10037, 10)
去除缺失的列后detail的形状为:  (10037, 9)
```

- 当 how 参数取值为 any 时，删除了一个特征，说明这个特征存在缺失值。
- 若 how 参数不取 any 这个默认值，而是取 all，则表示整个特征全部为缺失值时才会执行删除操作。

检测与处理缺失值

2. 替换法

- 替换法是指用一个特定的值替换缺失值。
- 特征可分为数值型和类别型，两者出现缺失值时的处理方法也是不同的。
  - 缺失值所在特征为**数值型**时，通常利用其均值、中位数和众数等描述其集中趋势的统计量来代替缺失值。
  - 缺失值所在特征为**类别型**时，则选择使用众数来替换缺失值。
- pandas库中提供了缺失值替换的方法：fillna，

检测与处理缺失值

2. 替换法

- pandas库中提供了缺失值替换的方法名为fillna，其基本语法如下。

*pandas.DataFrame.**fillna**(value=None, method=None, axis=None, inplace=False, limit=None)*

- 常用参数及其说明如下。

参数名称	说明
value	接收scalar，dict，Series或者DataFrame。表示用来替换缺失值的值。无默认。
method	接收特定string。Backfill 或 bfill 表示使用下一个非缺失值填补缺失值。 Pad 或 ffill 表示使用上一个非缺失值填补缺失值。默认为None。
axis	接收0或1。表示轴向。默认为1。
inplace	接收boolean。表示是否在原表上进行操作。默认为False。
limit	接收int。表示填补缺失值个数上限，超过则不进行填补。默认为None。

检测与处理缺失值

2. 替换法

```
detail = detail.fillna(-99)
print('detail每个特征缺失的数目为：\n',detail.isnull().sum())
```

```
In [15]: detail = detail.fillna(-99)
    ...: print('detail每个特征缺失的数目为: \n',detail.isnull().sum())
detail每个特征缺失的数目为:
 order_id             0
dishes_id             0
logicprn_name         0
dishes_name           0
itemis_add            0
counts                0
amounts               0
place_order_time      0
picture_file          0
emp_id                0
dtype: int64
```

检测与处理缺失值

3. 插值法

- 删除法简单易行，但是会引起数据结构变动，样本减少；替换法使用难度较低，但是会影响数据的标准差，导致信息量变动。在面对数据缺失问题时，除了这两种方法之外，还有一种常用的方法—插值法。
- 常用的插值法有线性插值、多项式插值和样条插值等：
  - 线性插值是一种较为简单的插值方法，它针对已知的值求出线性方程，通过求解线性方程得到缺失值。
  - 多项式插值是利用已知的值拟合一个多项式，使得现有的数据满足这个多项式，再利用这个多项式求解缺失值，常见的多项式插值法有拉格朗日插值和牛顿插值等。
  - 样条插值是以可变样条来作出一条经过一系列点的光滑曲线的插值方法，插值样条由一些多项式组成，每一个多项式都是由相邻两个数据点决定，这样可以保证两个相邻多项式及其导数在连接处连续。
- Pandas提供了对应的名为interploate的插值方法，能够进行上述部分插值操作，但是SciPy的interploate模块更加全面。

检测与处理缺失值

3. 插值法（线性插值）

```
import numpy as np
from scipy.interpolate import interp1d              ##一维数据插值函数
x=np.array([1,2,3,4,5,8,9,10])                      ##创建自变量x
y1=np.array([2,8,18,32,50,128,162,200])             ##创建因变量y1
y2=np.array([3,5,7,9,11,17,19,21])                  ##创建因变量y2
LinearInsValue1 = interp1d(x,y1,kind='linear')  ##线性插值拟合x,y1
LinearInsValue2 = interp1d(x,y2,kind='linear')   ##线性插值拟合x,y2
print('当x为6、7时，使用线性插值y1为：',LinearInsValue1([6,7]))
print('当x为6、7时，使用线性插值y2为：',LinearInsValue2([6,7]))
```

检测与处理缺失值

3. 插值法（线性插值）

```
In [10]: import numpy as np
    ...: from scipy.interpolate import interp1d
    ...: x=np.array([1,2,3,4,5,8,9,10]) ##创建自变量x
    ...: y1=np.array([2,8,18,32,50,128,162,200]) ##创建因变量y1
    ...: y2=np.array([3,5,7,9,11,17,19,21]) ##创建因变量y2
    ...: LinearInsValue1 = interp1d(x,y1,kind='linear') ##线性插值拟合x,y1
    ...: LinearInsValue2 = interp1d(x,y2,kind='linear') ##线性插值拟合x,y2
    ...: print('当x为6、7时，使用线性插值y1为: ',LinearInsValue1([6,7]))
    ...: print('当x为6、7时，使用线性插值y2为: ',LinearInsValue2([6,7]))
当x为6、7时，使用线性插值y1为:  [ 76. 102.]
当x为6、7时，使用线性插值y2为:  [13. 15.]
```

检测与处理缺失值

3. 插值法（拉格朗日插值）

```
from scipy.interpolate import lagrange ##拉格朗日插值
LargeInsValue1 = lagrange(x,y1)          ##拉格朗日插值拟合x,y1
LargeInsValue2 = lagrange(x,y2)          ##拉格朗日插值拟合x,y2
print('当x为6,7时，使用拉格朗日插值y1为：',LargeInsValue1([6,7]))
print('当x为6,7时，使用拉格朗日插值y2为：',LargeInsValue2([6,7]))
```

```
当x为6,7时，使用拉格朗日插值y1为：  [72. 98.]
当x为6,7时，使用拉格朗日插值y2为：  [13. 15.]
```

检测与处理缺失值

3. 插值法（样条插值）

```
from scipy.interpolate import spline
SplineInsValue1 = spline(x,y1,xnew=np.array([6,7]))      ##样条插值拟合x,y1
SplineInsValue2 = spline(x,y2,xnew=np.array([6,7]))      ##样条插值拟合x,y2
print('当x为6,7时，使用样条插值y1为：',SplineInsValue1)
print('当x为6,7时，使用样条插值y2为：',SplineInsValue2)
```

```
当x为6,7时，使用样条插值y1为： [72. 98.]
当x为6,7时，使用样条插值y2为： [13. 15.]
```

检测与处理缺失值

3. 插值法

```
x=np.array([1,2,3,4,5,8,9,10])                    ##创建自变量x
y1=np.array([2,8,18,32,50,128,162,200])           ##创建因变量y1
y2=np.array([3,5,7,9,11,17,19,21])                ##创建因变量y2
```

以上自变量x和因变量y1的关系为：$y1 = 2x^2$，自变量x和y2的关系为：$y2 = 2x+1$

- 从拟合结果可以看出多项式插值和样条插值在两种情况下拟合都非常出色，线性插值法只在自变量和因变量为线性关系的情况下拟合才较为出色。
- 而在实际分析过程中，自变量与因变量的关系是线性的情况非常少见，所以在大多数情况下，多项式插值和样条插值是较为合适的选择。
- SciPy库中的interpolate模块除了提供常规的插值法外，还提供了例如在图形学领域具有重要作用的重心坐标插值（BarycentricInterpolator）等。在实际应用中，需要根据不同的场景，选择合适的插值方法。

检测与处理异常值

异常值

- 异常值是指数据中个别值的数值明显偏离其余的数值，有时也称为离群点，检测异常值就是检验数据中是否有录入错误以及是否含有不合理的数据。
- 异常值的存在对数据分析十分危险，如果计算分析过程的数据有异常值，那么会对结果会产生不良影响，从而导致分析结果产生偏差乃至错误。
- 常用的异常值检测主要为 **3σ原则** 和 **箱线图分析** 两种方法。

检测与处理异常值

1. 3σ原则

- 3σ原则又称为拉依达法则。该法则就是先假设一组检测数据只含有随机误差，对原始数据进行计算处理得到标准差，然后按一定的概率确定一个区间，认为误差超过这个区间的就属于异常值。
- 这种判别处理方法仅适用于对正态或近似正态分布的样本数据进行处理，如下表所示，其中σ代表标准差，μ代表均值，x=μ为图形的对称轴。

数值分布	在数据中的占比
$(\mu-\sigma,\mu+\sigma)$	0.6827
$(\mu-2\sigma,\mu+2\sigma)$	0.9545
$(\mu-3\sigma,\mu+3\sigma)$	0.9973

数值分布	在数据中的占比
$(\mu-\sigma,\mu+\sigma)$	0.6827
$(\mu-2\sigma,\mu+2\sigma)$	0.9545
$(\mu-3\sigma,\mu+3\sigma)$	0.9973
数值分布	在数据中的占比
$(\mu-\sigma,\mu+\sigma)$	0.6827
$(\mu-2\sigma,\mu+2\sigma)$	0.9545
$(\mu-3\sigma,\mu+3\sigma)$	0.9973
数值分布	在数据中的占比
$(\mu-\sigma,\mu+\sigma)$	0.6827
$(\mu-2\sigma,\mu+2\sigma)$	0.9545
$(\mu-3\sigma,\mu+3\sigma)$	0.9973

数值分布	在数据中的占比
0.6827	0.6827
0.9545	0.9545
0.9973	0.9973

- 通过表格内容看出，数据的数值分布几乎全部集中在区间(μ-3σ,μ+3σ)内，超出这个范围的数据仅占不到0.3%。故根据小概率原理，可以认为超出3σ的部分数据为异常数据。

检测与处理异常值

1. 3σ原则

- 自行构建**3σ函数**，进行异常值识别

```
import pandas as pd
detail = pd.read_csv('data/detail.csv',index_col=0,encoding = 'gbk')
def outRange(Ser1):
    boolInd = (Ser1.mean()-3*Ser1.std()>Ser1) | (Ser1.mean()+3*Ser1. std()< Ser1) #判断是否在区间外
    index = np.arange(Ser1.shape[0])[boolInd]                #布尔索引，留下区间之内数据的index
    outrange = Ser1.iloc[index]
    return outrange
outlier = outRange(detail['counts'])
print('使用拉依达准则判定异常值个数为:',outlier.shape[0])
print('异常值的最大值为：',outlier.max())
print('异常值的最小值为：',outlier.min())
```

检测与处理异常值

1. 3σ原则

- 自行构建**3σ函数**，进行异常值识别

```
import pandas as pd
detail = pd.read_csv('data/detail.csv',index_col=0,encoding = 'gbk')
def outRange(Ser1):
    boolInd = (Ser1.mean()-3*Ser1.std()>Ser1) | (Ser1.mean()+3*Ser1. std()< Ser1) #判断是否在区间外
    index = np.arange(Ser1.shape[0])[boolInd]          #布尔索引，留下区间之内数据的index
    outrange = Ser1.iloc[index]
    return outrange
outlier = outRange(detail['counts'])
print('使用拉依达准则判定异常值个数为:',outlier.shape[0])
print('异常值的最大值为：',outlier.max())
print('异常值的最小值为：',outlier.min())
```

```
使用拉依达准则判定异常值个数为: 209
异常值的最大值为:  10
异常值的最小值为:  3
```

检测与处理异常值

2.箱线图分析

- 箱型图提供了识别异常值的一个标准，即异常值通常被定义为小于**QL-1.5IQR**或大于**QU+1.5IQR**的值。
  - QL称为下四分位数，表示全部观察值中有四分之一的数据取值比它小。
  - QU称为上四分位数，表示全部观察值中有四分之一的数据取值比它大。
  - IQR称为四分位数间距，是上四分位数QU与下四分位数QL之差，其间包含了全部观察值的一半。
- 箱线图依据实际数据绘制，真实、直观地表现出了数据分布的本来面貌，且没有对数据做任何限制性要求（不像3σ原则要求数据服从正态分布），其判断异常值的标准以四分位数和四分位数间距为基础。
- 四分位数给出了数据分布的中心、散布和形状的某种指示，具有一定的鲁棒性，即25%的数据可以变得任意远而不会很大地扰动四分位数，所以异常值通常不能对这个标准施加影响。鉴于此，箱线图识别异常值的结果比较客观，因此在识别异常值方面具有一定的优越性。

检测与处理异常值

2.箱线图分析

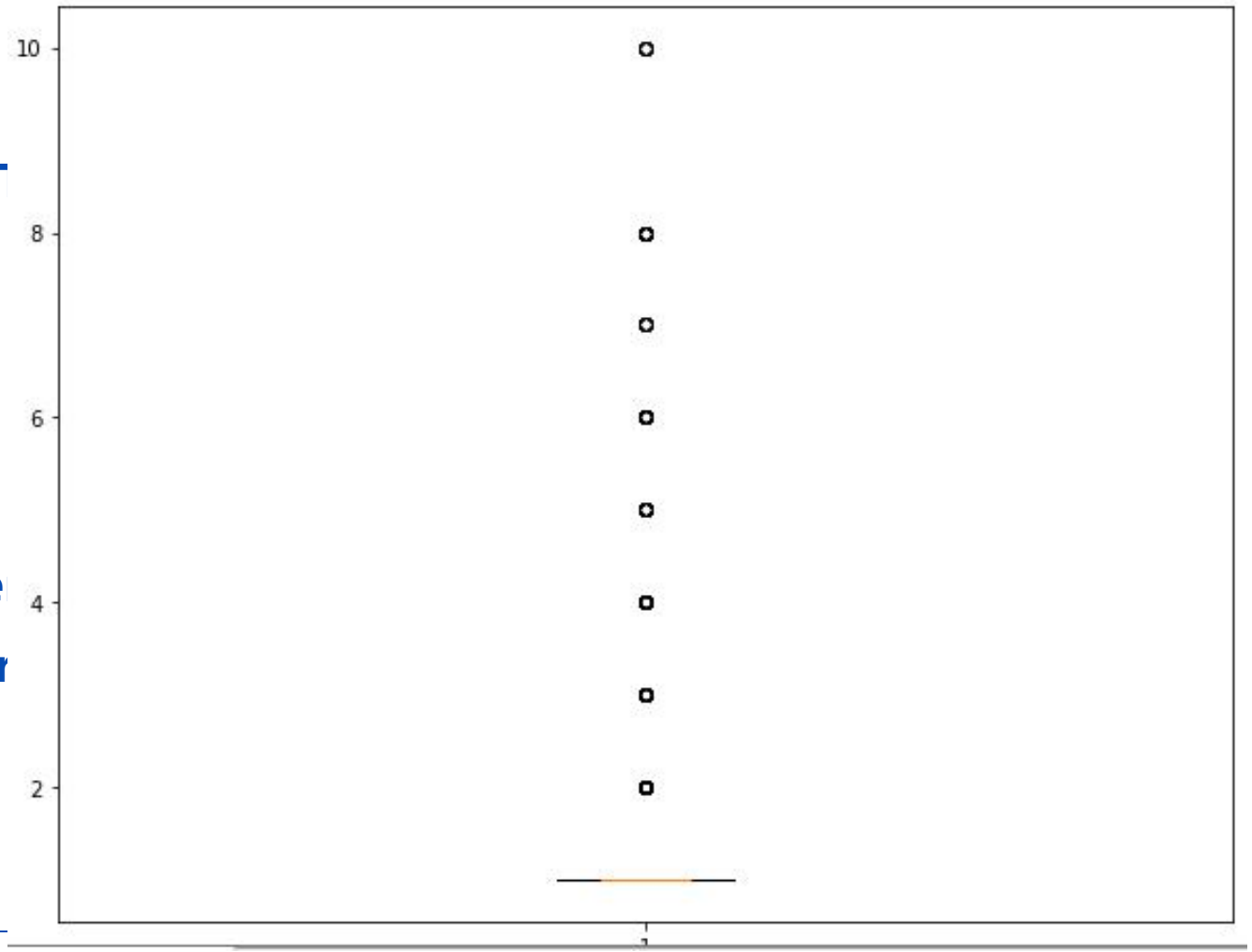
```
import matplotlib.pyplot as plt
plt.figure(figsize=(10,8))
p = plt.boxplot(detail['counts'].values,notch=True)        ##画出箱线图
outlier1 = p['fliers'][0].get_ydata()                      ##fliers为异常值的标签
plt.savefig('tmp/菜品异常数据识别.png')
plt.show()
print('销售量数据异常值个数为：',len(outlier1))
print('销售量数据异常值的最大值为：',max(outlier1))
print('销售量数据异常值的最小值为：',min(outlier1))
```

检测与处理异常值

2.箱线图分析

```
import matplotlib.pyplot as plt
plt.figure(figsize=(10,8))
p = plt.boxplot(detail['counts'].values,notch=
outlier1 = p['fliers'][0].get_ydata()
plt.savefig('tmp/菜品异常数据识别.png')
plt.show()
print('销售量数据异常值个数为：',len(outlier1))
print('销售量数据异常值的最大值为：',max(outlie
print('销售量数据异常值的最小值为：',min(outlier
```

```
销售量数据异常值个数为： 516
销售量数据异常值的最大值为： 10
销售量数据异常值的最小值为： 2
```



清洗数据任务实现

1.对订单详情表进行去重

- 菜品数据存在重复的现象，所以需要对菜品销售数据分别进行记录去重和特征去重。

2.处理订单详情表缺失值

- 统计经过去重操作后的各个特征缺失值，然后根据缺失情况选择缺失值处理方法。

3.处理菜品销售数据异常值

- 通过探索知道，在amount特征中存在异常值。使用替换法处理这些异常值。

清洗数据任务实现

1.对订单详情表进行去重

- 菜品数据存在重复的现象，所以需要对菜品销售数据分别进行记录去重和特征去重。

```
import pandas as pd
detail = pd.read_csv('data/detail.csv',index_col=0,encoding = 'gbk')
print('进行去重操作前订单详情表的形状为：',detail.shape)

##样本去重
detail.drop_duplicates(inplace = True)
print('进行去重操作后订单详情表的形状为：',detail.shape)
```

```
进行去重操作前订单详情表的形状为：  (10037, 18)
进行去重操作后订单详情表的形状为：  (10037, 18)
```

清洗数据任务实现

1.对订单详情表进行去重

```
##特征去重
def FeatureEquals(df):                ##定义求取特征是否完全相同的矩阵的函数
dfEquals=pd.DataFrame([],columns=df.columns,index=df.columns)
    for i in df.columns:
        for j in df.columns:
            dfEquals.loc[i,j]=df.loc[:,i].equals(df.loc[:,j])
    return dfEquals

detEquals=FeatureEquals(detail)          ## 应用上述函数
```

清洗数据任务实现

1.对订单详情表进行去重

```
##遍历所有数据，求取重复特征
lenDet = detEquals.shape[0]
dupCol = []
for k in range(lenDet):
    for l in range(k+1,lenDet):
        if detEquals.iloc[k,l] & (detEquals.columns[l] not in dupCol):
            dupCol.append(detEquals.columns[l])
##删除重复列
detail.drop(dupCol,axis=1,inplace=True)
print('进行去重操作后订单详情表的形状为：',detail.shape)
```

```
进行去重操作后订单详情表的形状为： (10037, 10)
```

清洗数据任务实现

2.处理订单详情表缺失值

- 统计经过去重操作后的各个特征缺失值，然后根据缺失情况选择缺失值处理方法。

```
naRate = (detail.isnull().sum()/detail.shape[0]*100).astype('str')+'%'
print('detail每个特征缺失的率为：\n',naRate)
##删除全部均为缺失的列
detail.dropna(axis = 1,how = 'all',inplace = True)
print('经过缺失值处理后订单详情表各特征缺失值的数目为：\n',detail.isnull().sum())
```

```
detail每个特征缺失的率为：
 order_id              0.0%
dishes_id              0.0%
logicprn_name        100.0%
dishes_name            0.0%
itemis_add             0.0%
counts                 0.0%
amounts                0.0%
place_order_time       0.0%
picture_file           0.0%
emp_id                 0.0%
```

```
经过缺失值处理后订单详情表各特征缺失值的数目为：
 order_id             0
dishes_id             0
dishes_name           0
itemis_add            0
counts                0
amounts               0
place_order_time      0
picture_file          0
emp_id                0
dtype: int64
```

清洗数据任务实现

3.处理菜品销售数据异常值

- 通过探索知道，在amount特征中存在异常值。使用替换法处理这些异常值。

```
##定义异常值识别与处理函数
def outRange(Ser1):
    QL = Ser1.quantile(0.25)        ##求出下四分位数
    QU = Ser1.quantile(0.75)        ##求出上四分位数
    IQR = QU-QL                     ##求出四分位数间距
    Ser1.loc[Ser1>(QU+1.5*IQR)] = QU    ##超出上限的异常值替换成上四分位数
    Ser1.loc[Ser1<(QL-1.5*IQR)] = QL    ##超出下限的异常值替换成下四分位数
    return Ser1
```

清洗数据任务实现

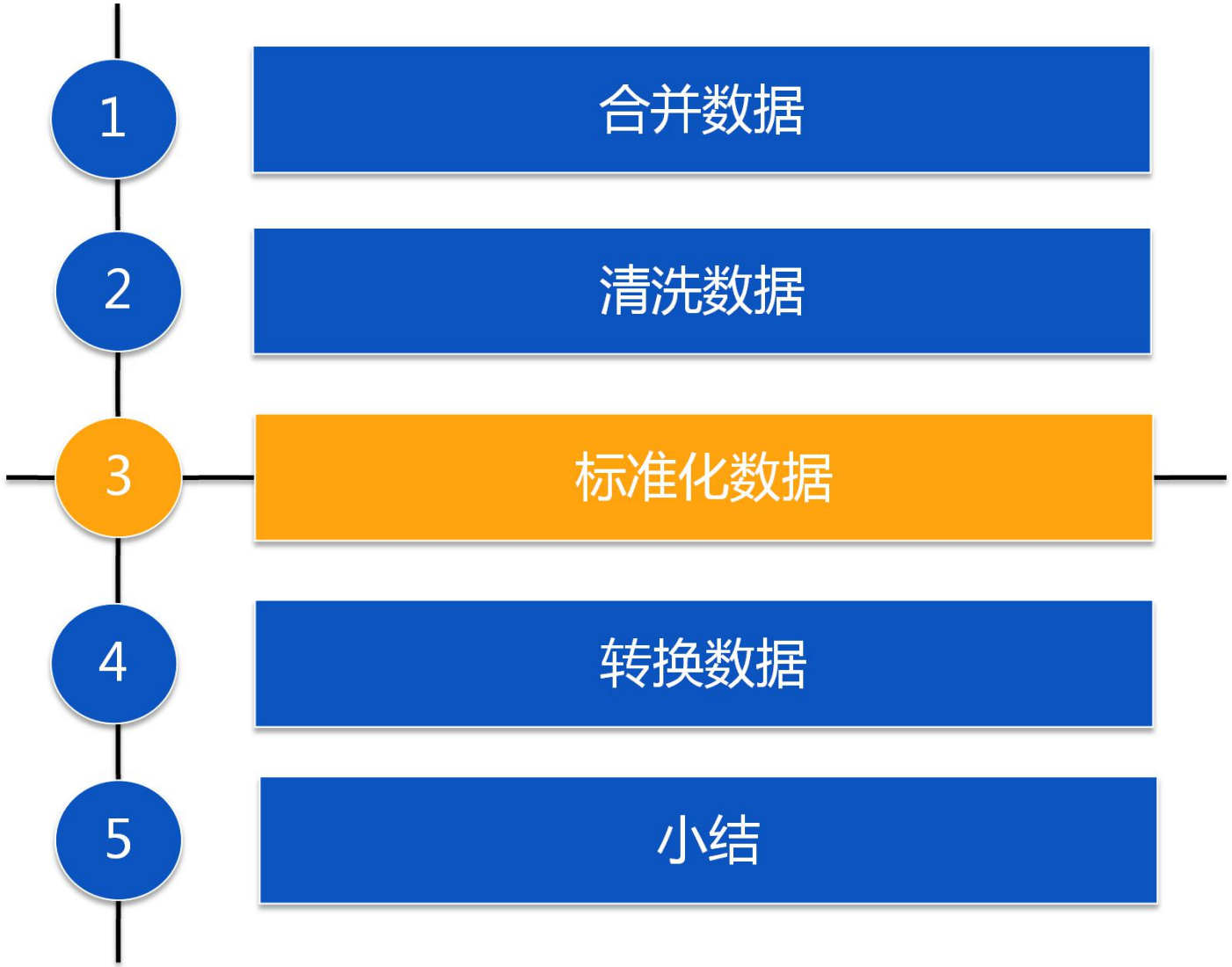
3.处理菜品销售数据异常值

- 通过探索知道，在amount特征中存在异常值。使用替换法处理这些异常值。

```
## 处理销售量和售价的异常值
detail['counts'] = outRange(detail['counts'])
detail['amounts'] = outRange(detail['amounts'])
##查看处理后的销售量和售价的最小值，最大值
print('销售量最小值为：', detail['counts'].min())
print('销售量最大值为：', detail['counts'].max())
print('售价最小值为：', detail['amounts'].min())
print('售价最大值为：', detail['amounts'].max())
```

```
销售量最小值为：  1.0
销售量最大值为：  1.0
售价最小值为：  1.0
售价最大值为：  99.0
```

目录



标准化数据

任务描述

- 不同特征之间往往具有不同的量纲，由此所造成的数值间的差异可能很大，在涉及空间距离计算或梯度下降法等情况时，不对其进行处理会影响到数据分析结果的准确性。为了消除特征之间量纲和取值范围差异可能造成的 影响，需要对数据进行标准化处理，也可以称作规范化处理。

任务分析

（1）离差标准化数据

（2）标准差标准化数据

（3）小数定标标准化数据

离差标准化数据

离差标准化公式

- 离差标准化是对原始数据的一种线性变换，结果是将原始数据的数值映射到[0,1]区间，转换公式为：

$$X^* = \frac{X - \min}{\max - \min}$$

- 其中，max为样本数据的最大值，min为样本数据的最小值，max-min为极差。离差标准化保留了原始数据值之间的联系，是消除量纲和数据取值范围影响最简单的方法。

离差标准化数据

离差标准化示例：离差标准化订单详情表中的销量和售价

```
import pandas as pd
import numpy as np
detail = pd.read_csv('data/detail.csv',index_col=0,encoding = 'gbk')
def MinMaxScale(data):                    ## 自定义离差标准化函数
    data=(data-data.min())/(data.max()-data.min())
    return data
data1=MinMaxScale(detail['counts'])    ##对菜品订单表销量做离差标准化
data2=MinMaxScale(detail ['amounts'])          ##对菜品订单表售价做离差标准化
data3=pd.concat([data1,data2],axis=1)
print('离差标准化之前销量和售价数据为：\n', detail[['counts','amounts']].head())
print('离差标准化之后销量和售价数据为：\n',data3.head())
```

离差标准化数据

离差标准化示例：离差标准化订单详情表中的销量和售价

```
def MinMaxScale(data):          ## 自定义离差标准化函数
    data=(data-data.min())/(data.max()-data.min())
    return data
data1=MinMaxScale(detail['counts'])    ##对菜品订单表销量做离差标准化
data2=MinMaxScale(detail ['amounts'])          ##对菜品订单表售价做离差标准化
data3=pd.concat([data1,data2],axis=1)
print('离差标准化之前销量和售价数据为：\n', detail[['counts','amounts']].head())
print('离差标准化之后销量和售价数据为：\n',data3.head())
```

离差标准化之前销量和售价数据为：

detail_id	counts	amounts
2956	1	49
2958	1	48
2961	1	30
2966	1	25
2968	1	13

离差标准化之后销量和售价数据为：

detail_id	counts	amounts
2956	0.0	0.271186
2958	0.0	0.265537
2961	0.0	0.163842
2966	0.0	0.135593
2968	0.0	0.067797

离差标准化数据

离差标准化的特点

- 通过代码运行，分析离差标准化前后的数据可以发现：
  - 数据的整体分布情况并不会随离差标准化而发生改变，原先取值较大的数据，在做完离差标准化后的值依旧较大。
  - 当数据和最小值相等的时候，通过离差标准化可以发现数据变为0。
  - 若数据极差过大就会出现数据在离差标准化后数据之间的差值非常小的情况。
- 同时，还可以看出离差标准化的缺点：
  - 若数据集中某个数值很大，则离差标准化的值就会接近于0，并且相互之间差别不大。若将来遇到超过目前属性[min,max]取值范围的时候，会引起系统出错，这时便需要重新确定min和max。

标准差标准化数据

标准差标准化的公式及特点

- 标准差标准化也叫零均值标准化或z分数标准化，是当前使用最广泛的数据标准化方法。经过该方法处理的数据均值为0，标准差为1，转化公式为：

$$X^* = \frac{X - \overline{X}}{\delta}$$

- 其中 $\overline{X}$ 为原始数据的均值， δ 为原始数据的标准差

标准差标准化数据

标准差标准化示例：标准差标准化订单详情表中的销量和售价

```
##自定义标准差标准化函数
def StandardScaler(data):
    data=(data-data.mean())/data.std()
    return data
##对菜品订单表售价和销量做标准化
data4=StandardScaler(detail['counts'])
data5=StandardScaler(detail['amounts'])
data6=pd.concat([data4,data5],axis=1)
print('标准差标准化之前销量和售价数据为：\n',detail[['counts','amounts']].head())
print('标准差标准化之后销量和售价数据为：\n',data6.head())
```

标准差标准化数据

标准差标准化示例：标准差标准化订单详情表中的销量和售价

```
def StandardScaler(data):          ##自定义标准差标准化函数
    data=(data-data.mean())/data.std()
    return data
data4=StandardScaler(detail['counts'])          ##对菜品订单表销量做标准化
data5=StandardScaler(detail['amounts'])         ##对菜品订单表售价做标准化
data6=pd.concat([data4,data5],axis=1)
print('标准差标准化之前销量和售价数据为：\n',detail[['counts','amounts']].head())
print('标准差标准化之后销量和售价数据为：\n',data6.head())
```

标准差标准化之前销量和售价数据为：

detail_id	counts	amounts
2956	1	49
2958	1	48
2961	1	30
2966	1	25
2968	1	13

标准差标准化之后销量和售价数据为：

detail_id	counts	amounts
2956	-0.177571	0.116671
2958	-0.177571	0.088751
2961	-0.177571	-0.413826
2966	-0.177571	-0.553431
2968	-0.177571	-0.888482

小数定标标准化数据

小数定标标准化公式及对比

- 通过移动数据的小数位数，将数据映射到区间[-1,1]，移动的小数位数取决于数据绝对值的最大值。转化公式：

$$X^* = \frac{X}{10^k}$$

小数定标标准化数据

小数定标标准化示例：小数定标标准化订单详情表中的销量和售价

```
##自定义小数定标标准化函数
def DecimalScaler(data):
    data=data/10**np.ceil(np.log10(data.abs().max()))
    return data
##对菜品订单表售价和销量做标准化
data7=DecimalScaler(detail['counts'])
data8=DecimalScaler(detail['amounts'])
data9=pd.concat([data7,data8],axis=1)
print('小数定标标准化之前的销量和售价数据：\n',detail[['counts','amounts']].head())
print('小数定标标准化之后的销量和售价数据：\n',data9.head())
```

小数定标标准化数据

小数定标标准化示例：小数定标标准化订单详情表中的销量和售价

```
def DecimalScaler(data):          ##自定义小数定标标准化函数
    data=data/10**np.ceil(np.log10(data.abs().max()))
    return data
data7=DecimalScaler(detail['counts'])          ##对菜品订单表销量做标准化
data8=DecimalScaler(detail['amounts'])         ##对菜品订单表售价做标准化
data9=pd.concat([data7,data8],axis=1)
print('小数定标标准化之前的销量和售价数据：\n',detail[['counts','amounts']].head())
print('小数定标标准化之后的销量和售价数据：\n',data9.head())
```

小数定标标准化之前的销量和售价数据：

detail_id	counts	amounts
2956	1	49
2958	1	48
2961	1	30
2966	1	25
2968	1	13

小数定标标准化之后的销量和售价数据：

detail_id	counts	amounts
2956	0.1	0.049
2958	0.1	0.048
2961	0.1	0.030
2966	0.1	0.025
2968	0.1	0.013

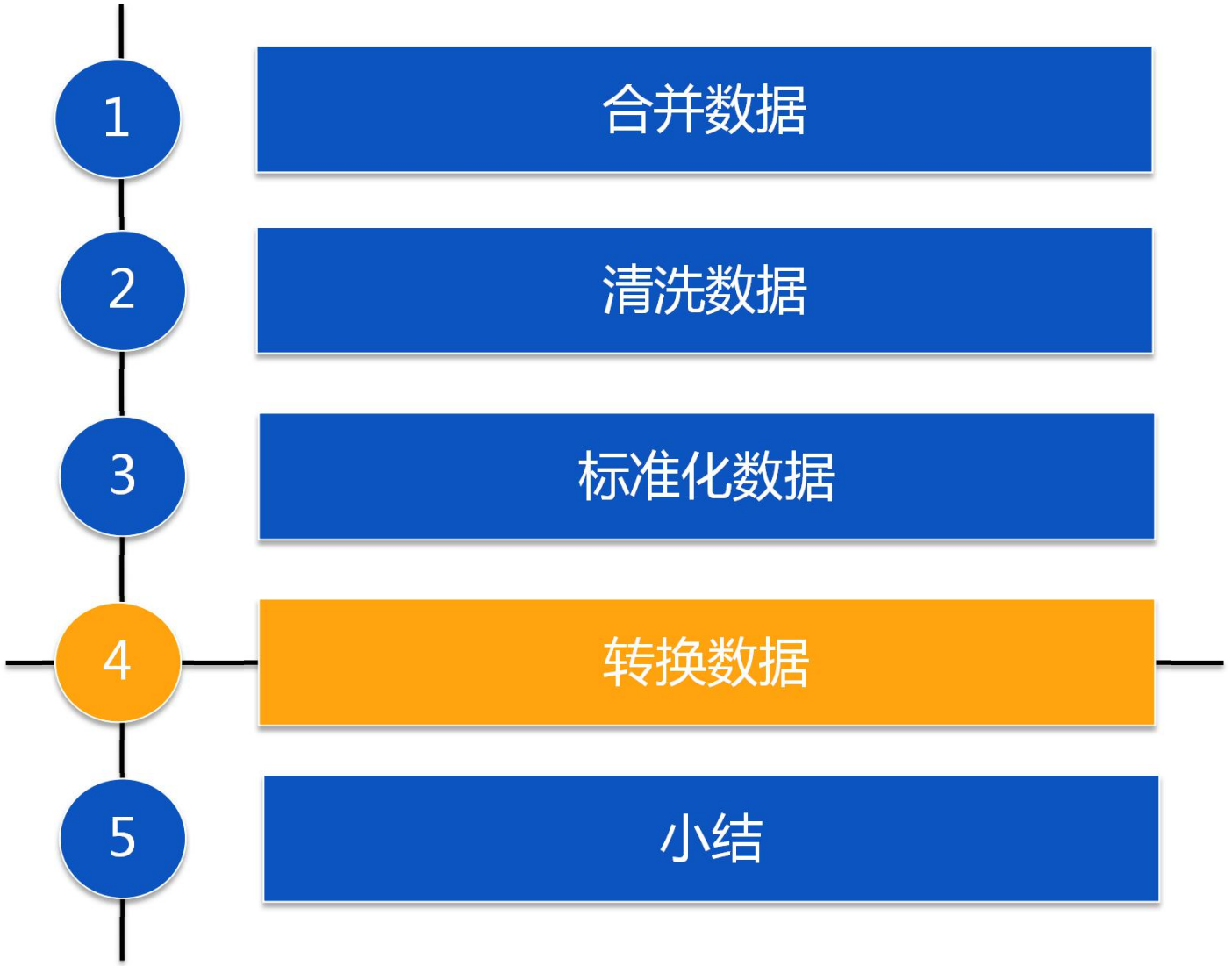
标准化数据

三种标准化方法对比

3种标准化方法各有其优势：

- 离线标准化方法简单，便于理解，标准化后的数据限定在[0,1]区间内。
- 标准差标准化受数据分布的影响最小。
- 小数定标标准化方法的适用范围广，并且受数据分布的影响小，相比较前两种方法，该方法使用程度适中。

目录



转换数据

任务描述

- 数据分析的预处理工作除了数据清理、数据合并以及数据标准化之外，还包括数据变换的过程。
- 数据即使经过了清洗、合并和标准化，依旧不能直接拿来做分析建模。
- 为了能够将数据分析工作继续往前推进，需要对数据做一些合理的转换，使之符合分析的要求。

哑变量处理类别数据

哑变量处理

- 数据分析模型中有相当一部分的算法模型都要求输入的特征为数值型，但实际数据中特征的类型不一定只有数值型，还会存在相当一部分的类别型，这部分的特征需要经过哑变量处理才可以放入模型之中。哑变量处理的原理示例如图。

哑变量处理前

	城市
1	广州
2	上海
3	杭州
4	北京
5	深圳
6	北京
7	上海
8	杭州
9	广州
10	深圳

哑变量处理后

	城市_广州	城市_上海	城市_杭州	城市_北京	城市_深圳
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
4	0	0	0	1	0
5	0	0	0	0	1
6	0	0	0	1	0
7	0	1	0	0	0
8	0	0	1	0	0
9	1	0	0	0	0
10	0	0	0	0	1

哑变量处理类别数据

get_dummies函数

- Python中可以利用pandas库中的get_dummies函数对类别型特征进行哑变量处理。

*pandas.**get_dummies**(data, prefix=None, prefix_sep='_', dummy_na=False, columns=None, sparse=False, drop_first=False)*

参数名称	说明
data	接收array、DataFrame或者Series。表示需要哑变量处理的数据。无默认。
prefix	接收string、string的列表或者string的dict。表示哑变量化后列名的前缀。默认为None。
prefix_sep	接收string。表示前缀的连接符。默认为‘_’。
dummy_na	接收boolean。表示是否为Nan值添加一列。默认为False。
columns	接收类似list的数据。表示DataFrame中需要编码的列名。默认为None，表示对所有object和category类型进行编码。
sparse	接收boolean。表示虚拟列是否是稀疏的。默认为False。
drop_first	接收boolean。表示是否通过从k个分类级别中删除第一级来获得k-1个分类级别。默认为False。

哑变量处理类别数据

get_dummies函数应用示例

- 菜品订单详情表中的菜品名称就是类别型特征，利用get_dummies函数对其进行哑变量处理。

```
import pandas as pd
import numpy as np
detail = pd.read_csv('data/detail.csv',encoding = 'gbk')
data=detail.loc[0:5,'dishes_name']        ##抽取部分数据做演示
print('哑变量处理前的数据为：\n',data)
print('哑变量处理后的数据为：\n',pd.get_dummies(data))
```

```
哑变量处理前的数据为：
 0      蒜蓉生蚝
1     蒙古烤羊腿
2      大蒜苋菜
3     芝麻烤紫菜
4       蒜香包
5       白斩鸡
Name: dishes_name, dtype: object
```

```
哑变量处理后的数据为：
   大蒜苋菜  白斩鸡  芝麻烤紫菜  蒙古烤羊腿  蒜蓉生蚝  蒜香包
0     0     0       0       0     1     0
1     0     0       0       1     0     0
2     1     0       0       0     0     0
3     0     0       1       0     0     0
4     0     0       0       0     0     1
5     0     1       0       0     0     0
```

哑变量处理类别数据

哑变量处理的特点

- 从代码运行结果看出，对于一个类别型特征，若其取值有m个，则经过哑变量处理后就变成了m个二元特征，并且这些特征互斥，每次只有一个激活，这使得数据变得稀疏。

总之：

- 对类别型特征进行哑变量处理主要解决了部分算法模型无法处理类别型数据的问题，这在一定程度上起到了扩充特征的作用。
- 由于数据变成了稀疏矩阵的形式，因此也加速了算法模型的运算速度。

离散化连续型数据

离散化

- 某些模型算法，特别是某些分类算法如ID3决策树算法和Apriori算法等，要求数据是离散的，此时就需要将连续型特征（数值型）变换成离散型特征（类别型）。
- 连续特征的离散化就是在数据的取值范围内设定若干个离散的划分点，将取值范围划分为一些离散化的区间，最后用不同的符号或整数值代表落在每个子区间中的数据值。
- 因此离散化涉及两个子任务，即确定分类数以及如何将连续型数据映射到这些类别型数据上。其原理如图。
- **常用的离散化方法主要有3种：等宽法、等频法、聚类分析法**

离散化处理前

	年龄
1	18
2	23
3	35
4	54
5	42
6	21
7	60
8	63
9	41
10	38

离散化处理后

	年龄
1	(17.955，27]
2	(17.955，27]
3	(27，36]
4	(45，54]
5	(36，45]
6	(17.955，27]
7	(54，63]
8	(54，63]
9	(36，45]
10	(36，45]

离散化连续型数据

1. 等宽法

- 将数据的值域分成具有相同宽度的区间，区间的个数由数据本身的特点决定或者用户指定，与制作频率分布表类似。pandas提供了cut函数，可以进行连续型数据的等宽离散化，其基础语法格式如下。

*pandas.**cut**(x, bins, right=True, labels=None, retbins=False, precision=3, include_lowest=False)*

参数名称	说明
x	接收数组或Series。代表需要进行离散化处理的数据。无默认。
bins	接收int，list，array，tuple。若为int，代表离散化后的类别数目；若为序列类型的数据，则表示进行切分的区间，每两个数间隔为一个区间。无默认。
right	接收boolean。代表右侧是否为闭区间。默认为True。
labels	接收list，array。代表离散化后各个类别的名称。默认为空。
retbins	接收boolean。代表是否返回区间标签。默认为False。
precision	接收int。显示的标签的精度。默认为3。

离散化连续型数据

1. 等宽法离散化示例

- 菜品售价amount使用cut函数进行等宽法离散化处理

```
price = pd.cut(detail['amounts'],5)
print('离散化后5条记录售价分布为：\n' ,price.value_counts())
```

```
In [13]: price = pd.cut(detail['amounts'],5)
    ...: print('离散化后5条记录售价分布为：\n' ,price.value_counts())
离散化后5条记录售价分布为：
 (0.823, 36.4]     5461
(36.4, 71.8]       3157
(71.8, 107.2]       839
(142.6, 178.0]      426
(107.2, 142.6]      154
Name: amounts, dtype: int64
```

离散化连续型数据

1. 等宽法

- 使用等宽法离散化的缺陷为：等宽法离散化对数据分布具有较高要求，若数据分布不均匀，那么各个类的数目也会变得非常不均匀，有些区间包含许多数据，而另外一些区间的数据极少，这会严重损坏所建立的模型。

离散化连续型数据

2. 等频法--示例

- cut函数虽然不能够直接实现等频离散化，但是可以通过定义将相同数量的记录放进每个区间。

```
def SameRateCut(data,k):
    w=data.quantile(np.arange(0,1+1.0/k,1.0/k))
    data=pd.cut(data,w)
    return data
```

```
In [18]: w
Out[18]:
0.0      1.0
0.2     18.0
0.4     32.0
0.6     39.0
0.8     58.0
1.0    178.0
```

```
result=SameRateCut(detail['amounts'],5).value_counts()  ##菜品售价等频法离散化
print('菜品数据等频法离散化后各个类别数目分布状况为：','\n',result)
```

```
菜品数据等频法离散化后各个类别数目分布状况为：
 (18.0, 32.0]     2107
(39.0, 58.0]     2080
(32.0, 39.0]     1910
(1.0, 18.0]      1891
(58.0, 178.0]    1863
Name: amounts, dtype: int64
```

离散化连续型数据

2. 等频法

- 等频法离散化的方法相比较于等宽法离散化而言，避免了类分布不均匀的问题，但同时却也有可能将数值非常接近的两个值分到不同的区间以满足每个区间中固定的数据个数。

离散化连续型数据

3. 基于聚类分析的方法

- 一维聚类的方法包括两个步骤：
  - 首先，将连续型数据用聚类算法（如K-Means算法等）进行聚类。
  - 然后，处理聚类得到的簇，将合并到一个簇的连续型数据做同一标记。
- 聚类分析的离散化方法需要用户指定簇的个数，用来决定产生的区间数。

离散化连续型数据

3. 基于聚类分析的离散化方法示例

```
def KmeanCut(data,k):              ##自定义数据k-Means聚类离散化函数
    from sklearn.cluster import KMeans              #引入KMeans
    kmodel=KMeans(n_clusters=k,n_jobs=4)        ##建立模型，n_jobs是并行数
    kmodel.fit(data.values.reshape((len(data), 1)))          ##训练模型
    c=pd.DataFrame(kmodel.cluster_centers_).sort_values(0)       ##输出聚类中心并排序
    w=c.rolling(2).mean.iloc[1:]              ##相邻两项求中点，作为边界点
    w=[0]+list(w[0])+[data.max()]             ##把首末边界点加上
    data=pd.cut(data,w)
    return data
result=KmeanCut(detail['amounts'],5).value_counts()    ##菜品售价聚类分析方法的离散化
print('菜品售价聚类离散化后各个类别数目分布状况为：','\n',result)
```

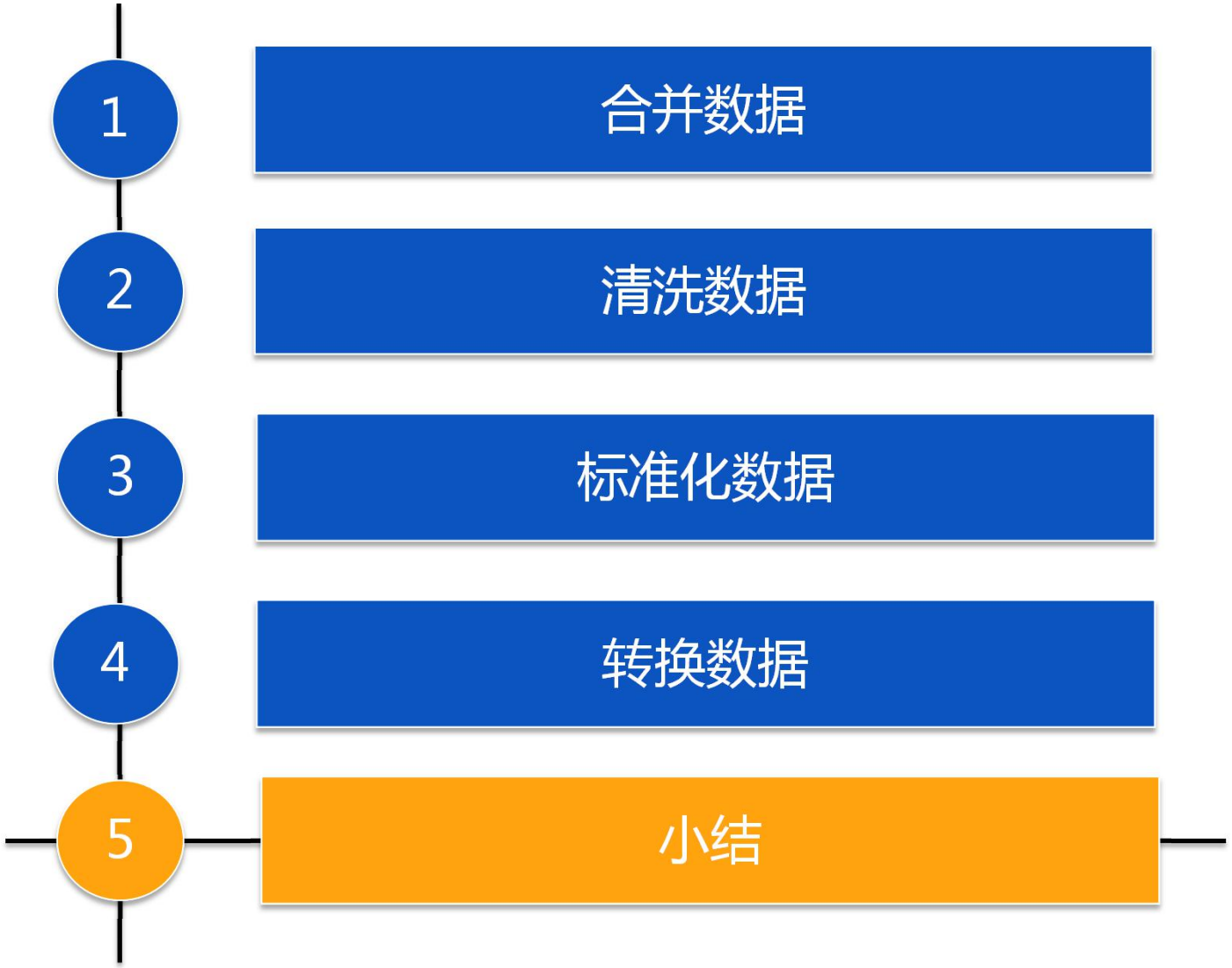
离散化连续型数据

3. 基于聚类分析的离散化方法示例

```
菜品售价聚类离散化后各个类别数目分布状况为：
 (22.31, 43.51]       3690
(43.51, 73.945]       2474
(0.0, 22.31]          2454
(73.945, 131.858]      993
(131.858, 178.0]       426
Name: amounts, dtype: int64
```

- k-Means聚类分析的离散化方法可以很好地根据现有特征的数据分布状况进行聚类，但是由于k-Means算法本身的缺陷，用该方法进行离散化时依旧需要指定离散化后类别的数目。此时需要配合聚类算法评价方法，找出最优的聚类簇数目。

目录



小结

本章以菜品数据为例子，实现了数据分析的数据预处理过程，即数据清洗、数据合并、数据标准化和数据转换。这四个步骤并不存在严格的先后关系，实际工作中往往需要交叉工作。

- **数据清洗**主要介绍了对重复数据、缺失值和异常值的处理。
  - 重复数据处理细分为记录去重和特征去重。
  - 缺失值处理方法分为删除、替换和插值。
  - 异常值介绍了3σ原则和箱线图识别这两种识别方法。
- **数据合并**是将多个数据源中的数据合并存放到一个数据存储的过程。
- **数据标准化**介绍了如何将不同量纲的数据转化为可以相互比较的标准化数据。
- **数据转换**介绍了如何从不同的应用角度对已有特征进行转换。

Thank you!