第3章

数据预处理

现实中获得的数据极易受到噪声、缺失值和不一致数据的影响。数据预处理是数据挖掘过程的第1个步骤，主要有数据清洗、数据集成、数据标准化、数据归约、数据变换和数据离散化等方法。

扫一扫

视频讲解

3.1 数据预处理的必要性

数据的质量决定了数据挖掘的效果，因此，在数据挖掘之前要对数据进行预处理以提高数据质量，从而改善数据挖掘的效果。

3.1.1 原始数据中存在的问题

现实世界中的数据大多都是“脏”的，主要存在以下问题。

1. 数据不一致

原始数据是从各种实际应用系统中获取的。由于各应用系统的数据缺乏统一的标准和定义，数据结构也有较大的差异，因此各系统间的数据存在严重的不一致性。例如，某数据库中不同表中的数据采用了不同的计量单位。

2. 噪声数据

收集数据时很难得到精确的数据，数据采集设备可能会出现故障，数据传输过程中可能会出现错误或存储介质损坏等情况，这些都会导致噪声数据的出现。

3. 缺失值

由于系统设计时可能存在缺陷或在系统使用过程中人为因素的影响，数据记录中可能出现有些数据属性值丢失或不确定的情况，从而造成数据的不完整。例如，数据采集传感器出现故障，导致一部分数据无法采集。

3.1.2　数据质量要求

数据挖掘需要的数据必须是高质量的数据，即数据挖掘所处理的数据必须具有准确性（Correctness）、完整性（Completeness）和一致性（Consistency）等性质。此外，时效性（Timeliness）、可信性（Believability）和可解释性（Interpretability）也会影响数据的质量。

1. 准确性

准确性是指数据记录的信息是否存在异常或错误。

2. 完整性

完整性是指数据信息是否存在缺失的情况。数据缺失可能是整条数据记录的缺失，也可能是数据中某个属性值的缺失。

3. 一致性

一致性是指数据是否遵循了统一的规范，数据集合是否保持了统一的格式。

4. 时效性

时效性是指某些数据是否能及时更新。更新时间越短，时效性越强。

5. 可信性

可信性是指用户信赖的数据的数量。用户信赖的数据越多，可信性越强。

6. 可解释性

可解释性是指数据自身是否易于理解。数据自身越容易被人理解，可解释性越强。

针对数据中存在的问题和数据质量要求，数据预处理过程主要包括数据清洗、数据集成、数据归约和数据变换等方法。

3.2　数据清洗

现实世界中的数据一般是不完整的、有噪声的或不一致的“脏”数据，无法直接进行数据挖掘或挖掘结果无法令人满意。数据清洗试图填充缺失的数据值、光滑噪声、识别离群点并纠正数据中的不一致。

扫一扫

视频讲解

3.2.1 数据清洗方法

1. 缺失值的处理

数据在收集和保存过程中，由于机械故障或人为的主观失误、历史局限或有意隐瞒等因素，会造成数据的缺失。对缺失值的处理主要有以下方法。

1）忽略元组

一般缺少类标号时直接忽略元组。除非元组有多个属性缺少值，否则该方法不是很有效。当每个属性缺失值的百分比变化很大时，它的性能特别差。采用忽略元组的方法会导致不能使用该元组的剩余属性值，而这些数据可能对数据分析是有用的。

2）人工填写缺失值

一般来说，该方法很费时，并且当数据集很大、缺失值很多时，该方法不太可行。

3）使用一个全局常量填充缺失值

将缺失的属性值用同一个常量（如 Unknown 或 $-\infty$）填充。如果缺失值都用 Unknown 替换，则挖掘程序可能误以为它们形成了一个有趣的概念，因为它们都具有相同的值。因此，尽管该方法简单，但是并不十分可靠。

4）使用属性的中心趋势度量（如均值或中位数）填充缺失值

中心趋势度量指示数据分布的“中间”值。对于正常的“对称”数据分布，可以使用均值；对于倾斜数据分布，应该使用中位数填充。例如，假定 All Electronics 的顾客收入的数据分布是对称的，并且平均收入为 56 000 美元，则使用该值替换 income 中的缺失值。

5）使用与给定元组属于同一类的所有样本的属性均值或中位数填充缺失值

例如，如果将顾客按 Credit Risk（信用风险）分类，则用具有相同信用风险的顾客的平均收入填充 income 中的缺失值。如果给定类的数据分布是倾斜的，中位数是更好的选择。

6）使用最可能的值填充缺失值

可以用回归、贝叶斯形式化方法等基于推理的工具或决策树归纳确定。例如，利用数据集中其他顾客的属性，可以构造一棵决策树预测缺失值。

2. 噪声数据的处理

噪声（Noise）是被测量的随机误差或方差。噪声的处理方法一般有分箱、回归和离群点分析等。

1）分箱

分箱（Binning）方法通过考查数据的“近邻”（即周围的值）光滑有序数据值。这些有序的数值被划分到一些“桶”或“箱”中。由于分箱方法考查近邻的值，因此它进行的是局部光滑。类似地，可以使用箱中数据的中位数光滑，此时，箱中的每个值都被替换为该箱中的中位数。对于用箱边界光滑，给定箱中的最大值和最小值同样被视为箱边界，而箱中的每个值都被替换为箱边界值。一般而言，宽度越大，光滑效果越明显。

2）回归

回归(Regression)用一个函数拟合数据光滑数据。线性回归涉及找出拟合两个属性(或变量)的“最佳”直线，使一个属性可以用来预测另一个。多元线性回归是线性回归的扩充，其中涉及的属性多于两个，并且数据被拟合到一个多维曲面。

3）离群点分析

离群点分析(Outlier Analysis)可以通过聚类等方法检测离群点。聚类将类似的值组织成群或簇。直观地，将落在簇集合之外的值视为离群点。

扫一扫

视频讲解

3.2.2 利用 Pandas 进行数据清洗

在许多数据分析工作中，经常会有缺失数据的情况。Pandas 的目标之一就是尽量轻松地处理缺失数据。

1. 检测与处理缺失值

Pandas 对象的所有描述性统计默认都不包括缺失数据。对于数值数据，Pandas 使用浮点值 NaN 表示缺失数据。

1）缺失值的检测与统计

isnull()函数可以直接判断该列中的哪个数据为 NaN。

【例 3-1】 利用 isnull()函数检测缺失值。

```
In[11]:    string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])
           print(string_data)
           string_data.isnull()
Out[11]:   0     aardvark
           1     artichoke
           2        NaN
           3      avocado
           dtype: object
           0    False
           1    False
           2    True
           3    False
           dtype: bool
```

在 Pandas 中，缺失值表示为 NA，它表示不可用(Not Available)。在统计应用中，NA 数据可能是不存在的数据，或者是存在却没有观察到的数据(如数据采集中发生了问题)。当清洗数据用于分析时，最好直接对缺失数据进行分析，以判断数据采集问题或缺失数据可能导致的偏差。Python 内置的 None 值也被当作 NA 处理。

【例 3-2】 Series 中的 None 值处理。

```
In[12]:    string_data = pd.Series(['aardvark', 'artichoke',np.nan, 'avocado'])
           string_data.isnull()
```

```
Out[12]:    0    False
            1    False
            2    True
            3    False
            dtype: bool
```

2）缺失值的统计

【例 3-3】 利用 isnull().sum()函数统计缺失值。

```
In[13]:    df = pd.DataFrame(np.arange(12).reshape(3,4),columns = ['A','B','C','D'])
           df.iloc[2,:] = np.nan
           df[3] = np.nan
           print(df)
           df.isnull().sum()
Out[13]:       A      B      C      D   3
           0  0.0    1.0    2.0    3.0  NaN
           1  4.0    5.0    6.0    7.0  NaN
           2  NaN    NaN    NaN    NaN  NaN
           A    1
           B    1
           C    1
           D    1
           3    3
           dtype: int64
```

另外，通过 info()函数也可以查看 DataFrame 每列数据的缺失情况。

【例 3-4】 利用 info()函数查看 DataFrame 的缺失值。

```
In[14]:    df.info()
Out[14]:   <class 'pandas.core.frame.DataFrame'>
           RangeIndex: 3 entries, 0 to 2
           Data columns (total 5 columns):
           A    2 non-null float64
           B    2 non-null float64
           C    2 non-null float64
           D    2 non-null float64
           3    0 non-null float64
           dtypes: float64(5)
           memory usage: 200.0 bytes
```

2. 缺失值的处理

1）删除缺失值

在缺失值的处理方法中，删除缺失值是常用的方法之一。通过 dropna()函数可以删除具有缺失值的行。

dropna()函数的语法格式为

```
dropna(axis = 0, how = 'any', thresh = None, subset = None, inplace = False)
```

dropna()函数的主要参数及其说明如表 3-1 所示。

表 3-1　dropna ()函数的主要参数及其说明

参　　数	说　　明
axis	默认为 axis＝0，当某行出现缺失值时，将该行丢弃并返回；当 axis＝1 时，为某列出现缺失值，将该列丢弃
how	确定缺失值个数，默认值 how＝'any'表明只要某行有缺失值就将该行丢弃；how＝'all'表明某行全部为缺失值才将其丢弃
thresh	阈值设定，行列中非默认值的数量小于给定的值，就将该行丢弃
subset	部分标签中删除某行列，如 subset＝['a'，'d']，即丢弃子列 a 和 d 中含有缺失值的行
inplace	布尔值，默认为 False，当 inplace＝True 时，即对原数据操作，无返回值

对于 Series，dropna()函数返回一个仅含非空数据和索引值的 Series。

【例 3-5】 Series 的 dropna()函数用法。

```
In[15]:    from numpy import nan as NA
           data = pd.Series([1, NA, 3.5, NA, 7])
           print(data)
           print(data.dropna())
Out[15]:   0    1.0
           1    NaN
           2    3.5
           3    NaN
           4    7.0
           dtype: float64
           0    1.0
           2    3.5
           4    7.0
           dtype: float64
```

当然，也可以通过布尔型索引达到这个目的。

【例 3-6】 布尔型索引选择过滤非缺失值。

```
In[16]:    not_null = data.notnull()
           print(not_null)
           print(data[not_null])
Out[16]:   0     True
           1    False
           2     True
           3    False
           4     True
           dtype: bool
           0    1.0
           2    3.5
           4    7.0
           dtype: float64
```

对于 DataFrame 对象，dropna()函数默认丢弃任何含有缺失值的行。

【例 3-7】 DataFrame 对象的 dropna()函数默认参数使用。

```
In[17]:    from numpy import nan as NA
           data = pd.DataFrame([[1., 5.5, 3.], [1., NA, NA],[NA, NA, NA],
           [NA, 5.5, 3.]])
           print(data)
           cleaned = data.dropna()
           print('删除缺失值后的: \n',cleaned)

Out[17]:       0    1    2
           0  1.0  5.5   3.0
           1  1.0  NaN   NaN
           2  NaN  NaN   NaN
           3  NaN  5.5   3.0
           删除缺失值后的:
              0    1    2
           0  1.0  5.5  3.0
```

传入 how='all'将只丢弃全为 NA 的那些行。

【例 3-8】 向 dropna()函数传入参数 how='all'。

```
In[18]:    data = pd.DataFrame([[1., 5.5, 3.], [1., NA, NA],[NA, NA, NA],
           [NA, 5.5, 3.]])
           print(data)
           print(data.dropna(how = 'all'))
Out[18]:       0    1    2
           0  1.0  5.5  3.0
           1  1.0  NaN  NaN
           2  NaN  NaN  NaN
           3  NaN  5.5  3.0

               0    1    2
           0  1.0  5.5  3.0
           1  1.0  NaN  NaN
           3  NaN  5.5  3.0
```

如果用同样的方式丢弃 DataFrame 的列，只需要传入 axis=1 即可。

【例 3-9】 dropna()函数中的 axis 参数应用。

```
In[19]:  data = pd.DataFrame([[1., 5.5, NA], [1., NA, NA],[NA, NA, NA], [NA, 5.5, NA]])
         print(data)
         print(data.dropna(axis = 1, how = 'all'))
Out[19]:     0    1    2
         0  1.0  5.5  NaN
         1  1.0  NaN  NaN
```

```
2  NaN  NaN  NaN
3  NaN  5.5  NaN

    0    1
0  1.0  5.5
1  1.0  NaN
2  NaN  NaN
3  NaN  5.5
```

可以使用 thresh 参数，当传入 thresh＝N 时，表示要求一行至少具有 N 个非 NaN 才能保留。

【例 3-10】 dropna()函数的 thresh 参数应用。

```
In[20]:    df = pd.DataFrame(np.random.randn(7, 3))
           df.iloc[:4, 1] = NA
           df.iloc[:2, 2] = NA
           print(df)
           print(df.dropna(thresh = 2))
Out[20]:              0          1             2
           0    0.176209        NaN         NaN
           1  - 0.871199        NaN         NaN
           2    1.624651        NaN    0.829676
           3  - 0.286038        NaN  - 1.809713
           4  - 0.640662    0.666998  - 0.032702
           5  - 0.453412  - 0.708945    1.043190
           6  - 0.040305  - 0.290658  - 0.089056

                      0          1           2
           2    1.624651        NaN    0.829676
           3  - 0.286038        NaN  - 1.809713
           4  - 0.640662    0.666998  - 0.032702
           5  - 0.453412  - 0.708945    1.043190
           6  - 0.040305  - 0.290658  - 0.089056
```

2）填充缺失值

直接删除缺失值并不是一个很好的方法，可以用一个特定的值替换缺失值。缺失值所在的特征为数值型时，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填充；缺失值所在特征为类别型数据时，则选择众数填充。Pandas 库中提供了缺失值替换的 fillna()函数。

fillna()函数的格式如下，主要参数及其说明如表 3-2 所示。

```
pandas.DataFrame.fillna(value = None, method = None, axsi = None, inplace = False, limit = None)
```

表 3-2 fillna()函数的主要参数及其说明

参　数	说　明
value	用于填充缺失值的标量值或字典对象
method	插值方式
axis	待填充的轴，默认 axis=0
inplace	修改调用者对象而不产生副本
limit	（对于前向和后向填充）可以连续填充的最大数量

通过对一个常数调用 fillna()函数，就会将缺失值替换为这个常数值，如 df.fillna(0)为用零代替缺失值。也可以对一个字典调用 fillna()函数，就可以实现对不同的列填充不同的值。

【例 3-11】 通过字典形式填充缺失值。

```
In[21]:    df = pd.DataFrame(np.random.randn(5,3))
           df.loc[:3,1] = NA
           df.loc[:2,2] = NA
           print(df)
           print(df.fillna({1:0.88,2:0.66}))
Out[21]:              0         1          2
           0    0.861692       NaN        NaN
           1    0.911292       NaN        NaN
           2    0.465258       NaN        NaN
           3   -0.797297       NaN  -0.342404
           4    0.658408  0.872754  -0.108814

                      0         1          2
           0    0.861692  0.880000   0.660000
           1    0.911292  0.880000   0.660000
           2    0.465258  0.880000   0.660000
           3   -0.797297  0.880000  -0.342404
           4    0.658408  0.872754  -0.108814
```

fillna()函数默认返回新对象，但也可以通过设置参数 inplace=True 对现有对象进行就地修改。对 reindex()函数有效的插值方法也可用于 fillna()函数。

【例 3-12】 fillna()函数中 method 参数的应用。

```
In[22]:    df = pd.DataFrame(np.random.randn(6, 3))
           df.iloc[2:, 1] = NA
           df.iloc[4:, 2] = NA
           print(df)
           print(df.fillna(method = 'fill'))
Out[22]:              0         1          2
           0   -1.180338  -0.663622   0.952264
           1   -0.219780  -1.356420   0.742720
           2   -2.169303       NaN   1.129426
```

```
3    0.139349         NaN  -1.463485
4    1.327619         NaN         NaN
5    0.834232         NaN         NaN

              0           1           2
0  -1.180338  -0.663622    0.952264
1  -0.219780  -1.356420    0.742720
2  -2.169303  -1.356420    1.129426
3    0.139349  -1.356420  -1.463485
4    1.327619  -1.356420  -1.463485
5    0.834232  -1.356420  -1.463485
```

可以利用 fillna()函数实现许多其他功能，如传入 Series 的均值或中位数。

【例 3-13】 用 Series 的均值填充。

```
In[23]:    data = pd.Series([1., NA, 3.5, NA, 7])
           data.fillna(data.mean())
Out[23]:   0    1.000000
           1    3.833333
           2    3.500000
           3    3.833333
           4    7.000000
           dtype: float64
```

【例 3-14】 DataFrame 中用均值填充。

```
In[24]:    df = pd.DataFrame(np.random.randn(4, 3))
           df.iloc[2:, 1] = NA
           df.iloc[3:, 2] = NA
           print(df)
           df[1] = df[1].fillna(df[1].mean())
           print(df)
Out[24]:              0         1           2
           0    0.656155  0.008442    0.025324
           1    0.160845  0.829127    1.065358
           2  -0.321155       NaN  -0.955008
           3    0.953510       NaN         NaN

                      0         1           2
           0    0.656155  0.008442    0.025324
           1    0.160845  0.829127    1.065358
           2  -0.321155  0.418785  -0.955008
           3    0.953510  0.418785         NaN
```

3. 数据值替换

数据值替换是将查询到的数据替换为指定数据。在 Pandas 中通过 replace()函数进行数据值的替换。

【例 3-15】 使用 replace()函数替换数据值。

```
In[25]:    data = {'姓名':['张三','小明','马芳','国志'],'性别':['0','1','0','1'],
           '籍贯':['北京','甘肃','','上海']}
           df = pd.DataFrame(data)
           df = df.replace('','不详')
           print(df)
Out[25]:     姓名  性别  籍贯
           0  张三    0   北京
           1  小明    1   甘肃
           2  马芳    0   不详
           3  国志    1   上海
```

也可以同时对不同值进行多值替换，参数传入的方式可以是列表，也可以是字典。传入的列表中，第 1 个列表为被替换的值，第 2 个列表为对应替换的值。

【例 3-16】 传入列表实现多值替换。

```
In[26]:    df = df.replace(['不详','甘肃'],['兰州','兰州'])
           print(df)
Out[26]:     姓名  性别  籍贯
           0  张三    0   北京
           1  小明    1   兰州
           2  马芳    0   兰州
           3  国志    1   上海
```

【例 3-17】 传入字典实现多值替换。

```
In[27]:    df = df.replace({'1':'男','0':'女'})
           print(df)
Out[27]:     姓名   性别  籍贯
           0  张三    女   北京
           1  小明    男   兰州
           2  马芳    女   兰州
           3  国志    男   上海
```

4. 利用函数或映射进行数据转换

在数据分析中，经常需要进行数据的映射或转换，在 Pandas 中可以自定义函数，然后通过 map()函数实现。

【例 3-18】 map()函数映射数据。

```
In[28]:    data = {'姓名':['张三','小明','马芳','国志'],'性别':['0','1','0','1'],
           '籍贯':['北京','兰州','兰州','上海']}
           df = pd.DataFrame(data)
```

```
In[28]:  df['成绩'] = [58,86,91,78]
         print(df)
         def grade(x):
             if x>=90:
                 return '优'
             elif 70<=x<90:
                 return '良'
             elif 60<=x<70:
                 return '中'
             else:
                 return '差'
         df['等级'] = df['成绩'].map(grade)
         print(df)
Out[28]:    姓名  性别  籍贯  成绩
         0  张三    0   北京   58
         1  小明    1   兰州   86
         2  马芳    0   兰州   91
         3  国志    1   上海   78

            姓名  性别  籍贯  成绩  等级
         0  张三    0   北京   58   差
         1  小明    1   兰州   86   良
         2  马芳    0   兰州   91   优
         3  国志    1   上海   78   良
```

5. 异常值检测

异常值是指数据中存在的数值明显偏离其余数据的值。异常值的存在会严重干扰数据分析的结果，因此经常要检验数据中是否有输入错误或含有不合理的数据。在数据统计方法中，一般常用散点图、箱线图和 3σ 法则检测异常值。

1）散点图

通过数据分布的散点图可以发现异常值。

【例 3-19】 利用散点图检测异常值。

```
In[29]:  wdf = pd.DataFrame(np.arange(20),columns = ['W'])
         wdf['Y'] = wdf['W']*1.5+2
         wdf.iloc[3,1] = 128
         wdf.iloc[18,1] = 150
         wdf.plot(kind = 'scatter',x = 'W',y = 'Y')
```

输出结果如图 3-1 所示。

2）箱线图

箱线图利用数据中的 5 个统计量（最小值、第一四分位数 Q_1、中位数 Q_2、第三四分位数 Q_3 和最大值）描述数据，它也可以粗略地体现数据是否具有对称性、分布的分散程度等信息。利用箱线图检测异常值时，将最大（最小）值设置为与四分位数 Q_3 和 Q_1 间距为 1.5 个 IQR（IQR $=Q_3-Q_1$）的值，即 min $=Q_1-1.5$IQR，max $=Q_3+1.5$IQR，小于 min 和大于 max 的值被认为是异常值。

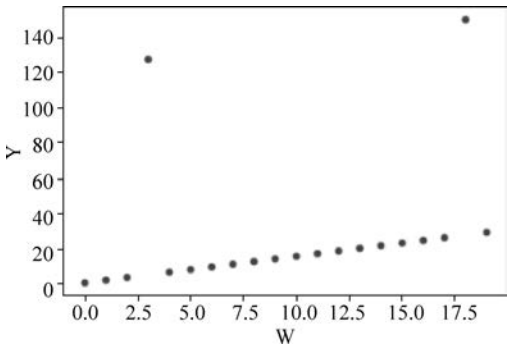

图 3-1　散点图检测异常值

【例 3-20】 利用箱线图分析异常值。

```
In[30]:    fig = wdf.boxplot(column = ['Y'],return_type = 'dict')
           min = fig['caps'][0].get_ydata()[0]      #获取下边缘值
           max = fig['caps'][1].get_ydata()[0]      #获取上边缘值
           print('max = ',max,'; min = ',min)
           print('离群点的索引是:')
           print(wdf[(wdf['Y']< min)|(wdf['Y']> max)].index)
Out[30]:   max = 30.5 ; min = 2.0
           离群点的索引是:
           Int64Index([3, 18], dtype = 'int64')
```

输出结果如图 3-2 所示。

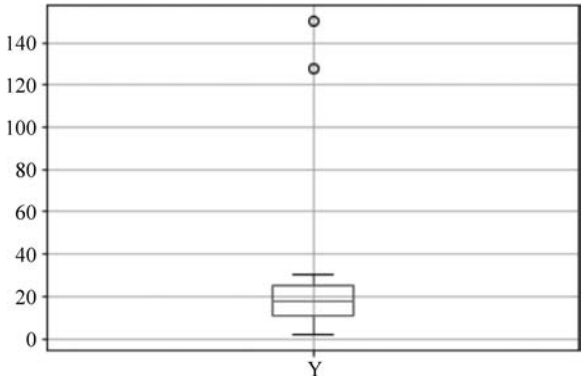

图 3-2　箱线图分析异常值

3）3σ 法则

如果数据服从正态分布，在 3σ 法则下，异常值被定义为一组测定值中与均值的偏差超过 3 倍标准差(σ)的值，因为在正态分布的假设下，距离均值 3σ 之外的值出现的概率小于 0.003。因此，根据小概率事件，可以认为超出 3σ 的值为异常数据。

【例 3-21】 利用 3σ 法则检测异常值。

```
In[31]:    def outRange(S):
               blidx = (S.mean() - 3 * S.std()> S)|(S.mean() + 3 * S.std()< S)
               idx = np.arange(S.shape[0])[blidx]
               outRange = S.iloc[idx]
               return outRange
           outier = outRange(wdf['Y'])
           outier
Out[31]:    18    150.0
            Name: Y, dtype: float64
```

3.3 数据集成

有时需要挖掘的数据可能来自多个数据源，导致数据存在冗余与不一致的情况，此时就需要对数据进行集成。数据集成是将多个数据源中的数据合并，存放于一个一致的数据存储中。

扫一扫

视频讲解

3.3.1 数据集成过程中的关键问题

1. 实体识别

实体识别问题是数据集成中的首要问题，因为来自多个信息源的现实世界的等价实体才能匹配。例如，在数据集成中如何判断一个数据库中的 customer_id 和另一数据库中的 cust_no 是指相同的属性？每个属性的元数据包含了属性名称、数据类型和属性的取值范围，因此，元数据可以用来避免模式集成的错误。在数据集成过程中，当一个数据库的属性与另一个数据库的属性匹配时，必须注意数据的结构，确保源系统中函数依赖和参数约束与目标系统中的匹配。

2. 数据冗余和相关分析

冗余是数据集成的另一个重要问题。如果一个属性能由另一个或另一组属性值“推导”，则这个属性可能是冗余的。属性命名不一致也会导致结果数据集中的冗余。有些冗余可以被相关分析检测到，对于标称属性，使用 χ^2（卡方）检验；对于数值属性，可以使用相关系数（Correlation Coefficient）和协方差（Covariance）评估属性间的相关性。

1）标称属性的 χ^2 检验

对于标称属性 A 和 B 之间的相关性，可以通过 χ^2 检验进行分析。假设 A 有 c 个不同的取值 $a_1,a_2,\cdots,a_c$，B 有 r 个不同的取值 $b_1,b_2,\cdots,b_r$。由 A 和 B 描述的数据元组可以用一个相依表显示，其中 A 的 c 个值构成列，B 的 r 个值构成行。令 (A_i,B_j) 表示属性 A 取 a_i，B 取 b_j 的联合事件，即 $(A=a_i,B=b_j)$。χ^2 值（又称为 Pearson χ^2 统计

量）的计算式为

$$\chi^2=\sum_{i=1}^{c}\sum_{j=1}^{r}\frac{(o_{ij}-e_{ij})^2}{e_{ij}} \tag{3-1}$$

其中，o_{ij} 为联合事件(A_i,B_j)的观测频度；e_{ij} 为(A_i,B_j)的期望频度，计算式为

$$e_{ij}=\frac{\text{count}(A=a_i)\times\text{count}(B=b_j)}{n} \tag{3-2}$$

其中，n 为数据元组个数；$\text{count}(A=a_i)$为 A 上具有值 a_i 的元组个数；$\text{count}(B=b_j)$为 B 上具有值 b_j 的元组个数。

2）数值属性的相关系数

对于数值数据，可以通过计算属性 A 和 B 的相关系数（又称为 Pearson 积矩系数）分析其相关性。相关系数 $r_{A,B}$ 定义为

$$r_{A,B}=\frac{\sum_{1}^{n}(a_i-\overline{A})(b_i-\overline{B})}{n\sigma_A\sigma_B}=\frac{\sum_{1}^{n}(a_ib_i)-n\overline{A}\,\overline{B}}{n\sigma_A\sigma_B} \tag{3-3}$$

其中，n 为元组个数；a_i 和 b_i 为元组 i 在 A 和 B 上的值；$\overline{A}$ 和 $\overline{B}$ 为 A 和 B 的均值；σ_A 和 σ_B 为 A 和 B 的标准差。$-1\leqslant r_{A,B}\leqslant 1$，如果相关系数 $r_{A,B}=0$，则 A 与 B 是独立的，它们之间不存在相关性；如果 $r_{A,B}<0$，则 A 与 B 负相关，一个值随另一个值减少而增加；如果 $r_{A,B}>0$，则 A 与 B 正相关，一个值随另一个值增加而增加，值越大，相关性越强。

3）数值属性的协方差

在概率论和统计学中，协方差用于衡量两个变量的总体误差。而方差是协方差的一种特殊情况，即当两个变量相同时的情况。

期望值分别为 $E[X]$和 $E[Y]$的两个随机变量 X 和 Y 之间的协方差 $\text{Cov}(X,Y)$定义为

$$\text{Cov}(X,Y)=E[(X-E(X))(Y-E(Y))]=E(XY)-E(X)E(Y) \tag{3-4}$$

如果两个变量的变化趋势一致，也就是说其中一个变量大于自身的期望值时，另一个变量也大于自身的期望值，那么两个变量之间的协方差就是正值；如果两个变量的变化趋势相反，即其中一个变量大于自身的期望值时，另一个变量却小于自身的期望值，那么两个变量之间的协方差就是负值。

【例 3-22】 利用 Python 计算属性间的相关性。

```
In[32]:    import pandas as pd
           import numpy as np
           a = [47, 83, 81, 18, 72, 41, 50, 66, 47, 20, 96, 21, 16, 60, 37, 59, 22, 16, 32, 63]
           b = [56, 96, 84, 21, 87, 67, 43, 64, 85, 67, 68, 64, 95, 58, 56, 75, 6, 11, 68, 63]
           data = np.array([a, b]).T
           dfab = pd.DataFrame(data, columns = ['A', 'B'])
```

```
                # display(dfab)
                print('属性 A 和 B 的协方差: ',dfab.A.cov(dfab.B))
                print('属性 A 和 B 的相关系数: ',dfab.A.corr(dfab.B))
Out[32]:        属性 A 和 B 的协方差: 310.2157894736842
                属性 A 和 B 的相关系数: 0.49924871046524394
```

3. 元组重复

除了检查属性的冗余之外，还要检测重复的元组，如给定唯一的数据实体，存在两个或多个相同元组的现象。

4. 数据值冲突检测与处理

数据集成还涉及数据值冲突检测与处理。例如，不同学校的学生交换信息时，由于不同学校有各自的课程计划和评分方案，同一门课的成绩采取的评分形式也有可能不同，如十分制或百分制，这样会使信息交换非常困难。

扫一扫

视频讲解

3.3.2　利用 Pandas 合并数据

在实际的数据分析中，可能有不同的数据来源，因此，需要对数据进行合并处理。

1. 使用 merge ()函数进行数据合并

Python 中的 merge()函数通过一个或多个键将两个 DataFrame 按行合并起来，与 SQL 中的 join 用法类似。Pandas 中的 merge()函数的语法格式为

```
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, validate = None)
```

merge()函数的主要参数及其说明如表 3-3 所示。

表 3-3　merge ()函数的主要参数及其说明

参　数	说　明
left	参与合并的左侧 DataFrame
right	参与合并的右侧 DataFrame
how	连接方法：inner，left，right，outer
on	用于连接的列名
left_on	左侧 DataFrame 中用于连接键的列
right_on	右侧 DataFrame 中用于连接键的列
left_index	左侧 DataFrame 中行索引作为连接键
right_index	右侧 DataFrame 中行索引作为连接键
sort	合并后会对数据排序，默认为 True
suffixes	修改重复名

【例 3-23】 merge()函数的默认合并数据。

```
In[33]:    price = pd.DataFrame({'fruit':['apple','grape',
           'orange','orange'],'price':[8,7,9,11]})
           amount = pd.DataFrame({'fruit':['apple','grape',
           'orange'],'amount':[5,11,8]})
           display(price,amount,pd.merge(price,amount))
```

输出结果如图 3-3 所示。

	fruit	price
0	apple	8
1	grape	7
2	orange	9
3	orange	11

	fruit	amount
0	apple	5
1	grape	11
2	orange	8

	fruit	price	amount
0	apple	8	5
1	grape	7	11
2	orange	9	8
3	orange	11	8

图 3-3 merge()函数的默认合并数据

由于两个 DataFrame 都有 fruit 列，所以默认按照该列进行合并，即 pd. merge (amount,price,on='fruit',how='inner')。如果两个 DataFrame 的列名不同，可以单独指定。

【例 3-24】 指定合并时的列名。

```
In[34]:    display(pd.merge(price,amount,left_on = 'fruit',right_on = 'fruit'))
```

输出结果如图 3-4 所示。

合并时默认使用内连接(inner)，即返回交集。通过设置 how 参数可以选择其他连接方法：左连接(left)、右连接(right)和外连接(outer)。

【例 3-25】 左连接。

```
In[35]:    display(pd.merge(price,amount,how = 'left'))
```

输出结果如图 3-5 所示。

【例 3-26】 右连接。

```
In[36]:    display(pd.merge(price,amount,how = 'right'))
```

输出结果如图 3-6 所示。

	fruit	price	amount
0	apple	8	5
1	grape	7	11
2	orange	9	8
3	orange	11	8

图 3-4　指定合并时的列名

	fruit	price	amount
0	apple	8	5
1	grape	7	11
2	orange	9	8
3	orange	11	8

图 3-5　左连接

	fruit	price	amount
0	apple	8	5
1	grape	7	11
2	orange	9	8
3	orange	11	8

图 3-6　右连接

也可以通过多个键进行合并。

【例 3-27】 通过多个键合并。

```
In[37]:    left = pd.DataFrame({'key1':['one','one','two'],
           'key2':['a','b','a'],'value1':range(3)})
           right = pd.DataFrame({'key1':['one','one','two','two'],
           'key2':['a','a','a','b'],'value2':range(4)})
           display(left,right,pd.merge(left,right,on = ['key1','key2'],how = 'left'))
```

输出结果如图 3-7 所示。

	key1	key2	value1
0	one	a	0
1	one	b	1
2	two	a	2

	key1	key2	value2
0	one	a	0
1	one	a	1
2	two	a	2
3	two	b	3

	key1	key2	value1	value2
0	one	a	0	0.0
1	one	a	0	1.0
2	one	b	1	NaN
3	two	a	2	2.0

图 3-7　通过多个键合并

在合并时会出现重复列名，虽然可以人为修改重复的列名，但 merge()函数提供了 suffixes 参数，用于处理该问题。

【例 3-28】 merge()函数中 suffixes 参数的应用。

```
In[38]:    print(pd.merge(left,right,on = 'key1'))
           print(pd.merge(left,right,on = 'key1',suffixes = ('_left','_right')))
Out[38]:     key1  key2_x  value1  key2_y  value2
           0  one      a       0       a       0
           1  one      a       0       a       1
           2  one      b       1       a       0
           3  one      b       1       a       1
```

```
4  two      a        2       a        2
5  two      a        2       b        3
   key1  key2_left value1 key2_right value2
0  one      a        0       a        0
1  one      a        0       a        1
2  one      b        1       a        0
3  one      b        1       a        1
4  two      a        2       a        2
5  two      a        2       b        3
```

2. 使用 concat ()函数进行数据连接

如果要合并的 DataFrame 之间没有连接键，就无法使用 merge()函数。Pandas 中的 concat()函数可以实现，默认情况下会按行的方向堆叠数据。如果要在列上连接，设置 axis＝1 即可。

【例 3-29】 两个 Series 的数据连接。

```
In[39]:   s1 = pd.Series([0,1],index = ['a','b'])
          s2 = pd.Series([2,3,4],index = ['a','d','e'])
          s3 = pd.Series([5,6],index = ['f','g'])
          print(pd.concat([s1,s2,s3]))            #Series 行合并
Out[39]:  a    0
          b    1
          a    2
          d    3
          e    4
          f    5
          g    6
          dtype: int64
```

【例 3-30】 两个 DataFrame 的数据连接。

```
In[40]:   data1 = pd.DataFrame(np.arange(6).reshape(2,3),columns = list('abc'))
          data2 = pd.DataFrame(np.arange(20,26).reshape(2,3),columns = list('ayz'))
          data = pd.concat([data1,data2],axis = 0)
          display(data1,data2,data)
```

输出结果如图 3-8 所示。

可以看出，连接方式为外连接(并集)，join 默认为"outer"，通过传入 join＝'inner'参数可以实现内连接。

	a	b	c
0	0	1	2
1	3	4	5

	a	y	z
0	20	21	22
1	23	24	25

	a	b	c	y	z
0	0	1.0	2.0	NaN	NaN
1	3	4.0	5.0	NaN	NaN
0	20	NaN	NaN	21.0	22.0
1	23	NaN	NaN	24.0	25.0

图 3-8　两个 DataFrame 的数据连接

【例 3-31】 指定索引顺序。

```
In[41]:    import pandas as pd
           s1 = pd.Series([0,1],index = ['a','b'])
           s2 = pd.Series([2,3,4],index = ['a','d','e'])
           s3 = pd.Series([5,6],index = ['f','g'])
           s4 = pd.concat([s1 * 5,s3],sort = False)
           s5 = pd.concat([s1,s4],axis = 1,sort = False)
           s6 = pd.concat([s1,s4],axis = 1,join = 'inner',sort = False)
           display(s5,s6)
```

输出结果如图 3-9 所示。

3. 使用 combine_first()函数合并数据

如果需要合并的两个 DataFrame 存在重复索引，则使用 merge()和 concat()函数都无法正确合并，此时需要使用 combine_first()函数。数据 w1 和 w2 如图 3-10 所示。

【例 3-32】 使用 combine_first()函数合并 w1 和 w2。

```
In[42]:    w1.combine_first(w2)
```

输出结果如图 3-11 所示。

	0	1
a	0.0	0
b	1.0	5
f	NaN	5
g	NaN	6

	0	1
a	0	0
b	1	5

图 3-9　指定索引顺序

	0	1
a	0	0
b	1	5

(a) w1

	0	1
a	0.0	0
b	1.0	5
f	NaN	5
g	NaN	6

(b) w2

图 3-10　数据 w1 和 w2

	0	1
a	0.0	0.0
b	1.0	5.0
f	NaN	5.0
g	NaN	6.0

图 3-11　使用 combine_first()函数合并 w1 和 w2

3.4 数据标准化

不同特征往往具有不同的量纲，由此造成数值间的差异很大。因此，为了消除特征之间量纲和取值范围的差异可能造成的影响，需要对数据进行标准化处理。

3.4.1 离差标准化数据

离差标准化是对原始数据所做的一种线性变换，将原始数据的数值映射到[0,1]区间，如式(3-5)所示。

$$x_1 = \frac{x - \min}{\max - \min} \tag{3-5}$$

【例 3-33】 数据的离差标准化。

```
In[43]:    def MinMaxScale(data):
               data = (data - data.min())/(data.max() - data.min())
               return data
           x = np.array([[ 1., -1.,  2.],[ 2.,  0.,  0.],[ 0.,  1., -1.]])
           print('原始数据为: \n',x)
           x_scaled = MinMaxScale(x)
           print('标准化后矩阵为:\n',x_scaled,end = '\n')
Out[43]:   原始数据为:
            [[ 1. -1.  2.]
            [ 2.  0.  0.]
            [ 0.  1. -1.]]
           标准化后矩阵为:
            [[0.66666667 0.         1.        ]
            [1.         0.33333333 0.33333333]
            [0.33333333 0.66666667 0.        ]]
```

3.4.2 标准差标准化数据

标准差标准化又称为零均值标准化或 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，如式(3-6)所示。

$$x_1 = \frac{x - \text{mean}}{\text{std}} \tag{3-6}$$

【例 3-34】 数据的标准差标准化。

```
In[44]:    def StandardScale(data):
               data = (data - data.mean())/data.std()
               return data
           x = np.array([[ 1., -1.,  2.],[ 2., 0., 0.],[ 0., 1., -1.]])
           print('原始数据为: \n',x)
```

```
x_scaled = StandardScale(x)
print('标准化后矩阵为:\n',x_scaled,end = '\n')
Out[44]:    原始数据为:
 [[ 1.  -1.   2.]
 [ 2.   0.   0.]
 [ 0.   1.  -1.]]
标准化后矩阵为:
 [[ 0.52128604 -1.35534369  1.4596009 ]
 [ 1.4596009   -0.41702883 -0.41702883]
 [-0.41702883   0.52128604 -1.35534369]]
```

数据归一化/标准化的目的是获得某种“无关性”，如偏置无关、尺度无关、长度无关等。当归一化/标准化方法背后的物理意义和几何含义与当前问题的需要相契合时，会对解决该问题有正向作用，反之则会起反作用。因此，如何选择标准化方法取决于待解决的问题。一般来说，涉及或隐含距离计算以及损失函数中含有正则项的算法，如 *K*-Means、KNN、PCA、SVM 等，需要进行数据标准化；距离计算无关的概率模型和树模型，如朴素贝叶斯、决策树和随机森林等，则不需要进行数据标准化。

扫一扫

视频讲解

3.5　数据归约

现实中数据集可能会很大，在海量数据集上进行数据挖掘需要很长的时间，因此要对数据进行归约。数据归约(Data Reduction)是指在尽可能保持数据完整性的基础上得到数据的归约表示。也就是说，在归约后的数据集上挖掘将更有效，而且仍会产生相同或相似的分析结果。数据归约包括维归约、数量归约和数据压缩。

3.5.1　维归约

维归约(Dimensionality Reduction)的思路是减少所考虑的随机变量或属性的个数，使用的方法有属性子集选择、小波变换和主成分分析。属性子集选择是一种维归约方法，其中不相关、弱相关或冗余的属性(或维)被检测或删除；后两种方法是将原始数据变换或投影到较小的空间。

1. 属性子集选择

属性子集选择通过删除不相关或冗余的属性(或维)减少数据量。属性选择的目的是找出最小属性集，使数据类的概率分布尽可能接近使用所有属性得到的原分布。在缩小的属性集上挖掘可以减少出现在发现模式上的属性数目，使模式容易理解。

如何找出原来属性的一个“好的”子集？对于 n 个属性，有 2^n 个可能的子集。穷举搜索找出最佳子集是不现实的。因此，通常使用压缩搜索空间的启发式算法进行“最佳”子集选取。它的策略是做局部最优选择，期望由此导出全局最优解。基本启发式方法包括以下技术。

1）逐步向前选择

逐步向前选择过程由空属性集作为归约集的起始，确定原属性集中最好的属性并添加到归约集中，迭代将剩余的原属性集中最好的属性添加到该集合中。

2）逐步向后删除

逐步向后删除过程由整个属性集开始，在每次迭代中删除尚在属性集中最差的属性。

3）逐步向前选择和逐步向后删除的结合

该方法将逐步向前选择和逐步向后删除相结合，每步选择一个最好的属性，并在属性中删除一个最差的属性。

4）决策树归纳

决策树算法构造一个类似于流程图的结构，每个内部节点表示一个属性上的测试，每个分支对应测试的一个结果。在每个节点上选择“最好”的属性，将数据划分成类。利用决策树进行子集选择时，由给定的数据构造决策树，不出现在树中的所有属性假定是不相关的，出现在树中的属性形成归约后的属性子集。

这些方法的结束条件可以不同，可以使用一个度量阈值决定何时终止属性选择过程。

在有些情况下，可以基于已有属性构造一些新属性，以提高准确性和对高维数据结构的理解，如根据已有的属性“高度”和“宽度”构造新属性“面积”。通过组合属性，属性构造可以发现关于数据属性间联系的缺失信息。

2. 小波变换

小波变换是一种新的变换分析方法，它继承和发展了短时傅里叶变换局部化的思想，同时又克服了窗口大小不随频率变化等缺点，能够提供一个随频率改变的时间-频率窗口，是进行信号时频分析和处理的理想工具。对随机信号进行小波变换可以得到与原数据长度相等的频域系数，由于在频域，信号能量主要集中在低频，因此可以截取中低频的系数保留近似的压缩数据。

【例 3-35】 对图像进行小波变换并显示。

```
In[45]:    import numpy as np
           import pywt
           import cv2 as cv
           import matplotlib.pyplot as plt
           img = cv.imread("lena_color_256.tif")
           img = cv.resize(img, (448, 448))
           # 将多通道图像转换为单通道图像
           img = cv.cvtColor(img, cv2.COLOR_BGR2GRAY).astype(np.float32)
           plt.figure('二维小波一级变换')
           coeffs = pywt.dwt2(img, 'haar')
           cA, (cH, cV, cD) = coeffs
           # 将各子图进行拼接,最后得到一幅图
           AH = np.concatenate([cA, cH + 255], axis = 1)
```

```
VD = np.concatenate([cV + 255, cD + 255], axis = 1)
img = np.concatenate([AH, VD], axis = 0)
# 显示为灰度图
plt.axis('off')
plt.imshow(img,'gray')
plt.title('result')
plt.show()
```

输出结果如图 3-12 所示。

图 3-12　图像的小波变换

3. 主成分分析

1）算法原理

主成分分析（PCA）又称为 Karhunen-Loeve 或 K-L 方法，用于搜索 k 个最能代表数据的 n 维正交向量，是最常用的一种降维方法。PCA 通常用于高维数据集的探索与可视化，还可以用作数据压缩和预处理等，在数据压缩消除冗余和数据噪声消除等领域也有广泛的应用。

PCA 的主要目的是找出数据中最主要的方面代替原始数据。具体地，假如数据集是 n 维的，共有 m 个数据（$x(1),x(2),\cdots,x(m)$），希望将这 m 个数据从 n 维降到 n' 维，使这 m 个 n' 维数据集尽可能代表原始数据集。

2）PCA 算法

输入：n 维样本集 $D=(x(1),x(2),\cdots,x(m))$，降维后的维数 n'

输出：降维后的样本集 D'

方法：

(1) 对所有的样本进行中心化：$x(i)'=x(i)-\frac{1}{m}\sum_{j=1}^{m}x(j)$；

(2) 计算样本的协方差矩阵 $\boldsymbol{XX}^{\mathrm{T}}$；

(3) 对协方差矩阵进行特征值分解；

(4) 取出最大的 n'个特征值对应的特征向量（$w_1,w_2,\cdots,w_{n'}$），将所有特征向量标准化后，组成特征向量矩阵 $\boldsymbol{W}$；

(5) 将样本集中的每个样本 $x(i)$转换为新的样本 $z(i)=\boldsymbol{W}^{\mathrm{T}}x(i)$；

(6) 得到输出样本集 $D'=(z(1),z(2),\cdots,z(m))$。

【例 3-36】 sklearn 实现鸢尾花数据进行降维，将原来 4 维的数据降维为二维。

```
In[46]:    import matplotlib.pyplot as plt
           from sklearn.decomposition import PCA
           from sklearn.datasets import load_iris
           data = load_iris()
           y = data.target
           x = data.data
           pca = PCA(n_components = 2)
```

```
#加载 PCA 算法,设置降维后主成分数目为 2
reduced_x = pca.fit_transform(x)                    #对样本进行降维
#在平面中画出降维后的样本点分布
red_x,red_y = [],[]
blue_x,blue_y = [],[]
green_x,green_y = [],[]
for i in range(len(reduced_x)):
    if y[i] == 0:
        red_x.append(reduced_x[i][0])
        red_y.append(reduced_x[i][1])
    elif y[i] == 1:
        blue_x.append(reduced_x[i][0])
        blue_y.append(reduced_x[i][1])
    else:
        green_x.append(reduced_x[i][0])
        green_y.append(reduced_x[i][1])
plt.scatter(red_x,red_y,c = 'r',marker = 'x')
plt.scatter(blue_x,blue_y,c = 'b',marker = 'D')
plt.scatter(green_x,green_y,c = 'g',marker = '.')
plt.show()
```

降维后的样本点分布如图 3-13 所示。

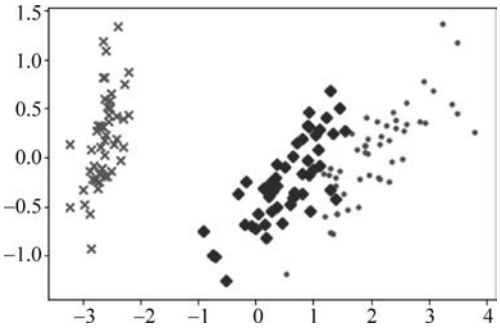

图 3-13　降维后的样本点分布

3.5.2　数量归约

数量归约(Numerosity Reduction)是指用替代的、较小的数据表示形式转换原始数据。这些方法可以是参数或非参数的。参数方法使用模型估计数据,使得一般只需要存放模型参数而不是实际数据(离群点须存放),如回归和对数线性模型；非参数方法包括直方图、聚类、抽样和数据立方体聚类。

1. 回归和对数线性模型

回归和对数线性模型可以用来近似给定的数据。在(简单)线性模型中,对数据拟合得到一条直线,多元回归是(简单)线性回归的扩展,用两个或多个自变量的线性函数对

因变量建模。

对数线性模型近似离散的多维概率分布，是用于离散型数据或整理成列联表格式的计数资料的统计分析工具。给定 n 维元组的集合，把每个元组看作 n 维空间中的点。对于离散属性集，可以使用对数线性模型基于维组合的一个较小子集估计多维空间中每个点的概论。因此，将高维数据空间由较低维空间构造。

2. 直方图

直方图使用分箱思路近似数据分布。用直方图归约数据，就是将直方图中的桶的个数由观测值的数量 n 减少到 k，使数据变成一块一块地呈现。为了压缩数据，通常让一个桶代表给定属性的一个连续值域。桶的划分可以是等宽的，也可以是等频的。

3. 聚类

聚类算法是将数据划分为簇，使簇内的数据对象尽可能“相似”，而簇间的数据对象尽可能“相异”。在数据归约中，用每个数据簇中的代表替换实际数据，以达到数据归约的效果。

4. 抽样

抽样通过选取随机样本(子集)，实现用小数据代表大数据的过程。抽样的方法包括简单随机抽样、簇抽样和分层抽样等。

5. 数据立方体聚类

数据立方体聚类用于从低粒度的数据分析聚合成汇总粒度的数据分析。一般认为最细的粒度是一个最小的立方体，在此之上每个高层次的抽象都能形成一个更大的立方体。数据立方体聚类就是将细粒度的属性聚集到粗粒度的属性。

3.5.3 数据压缩

数据压缩(Data Compression)使用变换，以便得到原始数据的归约或“压缩”表示。如果数据可以在压缩后重构，而不损失信息，则该数据归约称为无损的；如果是近似重构原数据，则称为有损的。基于小波变换的数据压缩是一种非常重要的有损压缩方法。

小波变换(Wavelet Transform)是 20 世纪 80 年代后期发展起来的继傅里叶变换用于信息处理与分析的强大工具。对数据向量 $\boldsymbol{X}$ 进行小波变换，会得到具有相同长度的小波系数；对小波系数进行小波逆变换，会还原数据向量 $\boldsymbol{X}$。由于数据的主要能量集中在低频区域，因此仅存储一小部分最强(低频部分)的小波系数，就能保留近似的压缩数据。给定一组系数，使用相应的小波逆变换可以构造原数据的近似。

3.6 数据变换与数据离散化

扫一扫

视频讲解

在数据预处理过程中，不同的数据适合不同的数据挖掘算法。数据变换是一种将原

始数据变换成较好数据格式的方法，以便作为数据处理前特定数据挖掘算法的输入。数据离散化是一种数据变换的形式。

3.6.1 数据变换的策略

在数据变换中，数据被变换成适合数据挖掘的形式。数据变换主要有以下几种方法。

1. 光滑

数据光滑用于去除数据中的噪声。常用的数据光滑方法有分箱、回归和聚类等。

2. 属性构造

属性构造是通过给定的属性构造新的属性并添加到属性集中，以帮助数据挖掘。

3. 聚集

聚集是对数据进行汇总或聚集。例如，可以聚集日销售数据，计算月和年销售量。通常，聚集用来为多个抽象层的数据分析构造数据立方体。

4. 规范化

把属性数据按比例缩放，使之落入一个特定的小区间，如[-1.0,1.0]。

1）最小-最大规范化

最小-最大规范化对原始数据进行线性变换。假设 $\min_A$ 和 $\max_A$ 分别为属性 A 的最小值和最大值。最小-最大规范化的计算式为

$$v'_i=\frac{v_i-\min_A}{\max_A-\min_A}(\text{new\_max}_A-\text{new\_min}_A)+\text{new\_min}_A \tag{3-7}$$

把属性 A 的值 v_i 映射到[new_min_A,new_max_A]中的 v'_i。最小-最大规范化保持原始数据值之间的联系。如果输入实例落在原数据值域之外，则该方法将面临“越界”错误。

2）z 分数规范化

在 z 分数(z-score)规范化(或零均值规范化)中，属性 A 的值 v_i 映射为 v'_i的计算式为

$$v'_i=\frac{v_i-\overline{A}}{\sigma_A} \tag{3-8}$$

其中，$\overline{A}$ 和 σ_A 分别为属性 A 的均值和标准差。

3）小数定标

小数定标规范化通过移动属性 A 的值的小数点位置进行规范化。小数点的移动位数依赖于 A 的最大绝对值。属性 A 的值 v_i 映射为 v'_i的计算式为

$$v'_i=\frac{v_i}{10^j} \tag{3-9}$$

其中，j 为使 $\max(|v_i'|)<1$ 的最小整数。

5. 离散化

数值属性(如年龄)的原始值用区间标签(如 0～10、11～20 等)或概念标签(如青年、中年、老年)替换。这些标签可以递归地组织成更高层概念，形成数值属性的概念分层，以适应不同用户的需要。

1) 通过分箱离散化

分箱是一种基于指定的箱个数的自顶向下的分裂技术。例如，使用等宽或等频分箱，然后用箱均值或中位数替换箱中的每个值，可以将属性值离散化。分箱对用户指定的箱个数很敏感，也易受离群点的影响。

2) 通过直方图离散化

直方图把属性 A 的值划分为不相交的区间，称为桶或箱。可以使用各种划分规则定义直方图。例如，在等宽直方图中，将值分成相等分区或区间。直方图分析算法可以递归地用于每个分区，自动地产生多级概念分层，直到达到一个预先设定的概念层数，过程终止。

3) 通过聚类、决策树和相关性分析离散化

聚类、决策树和相关性分析可以用于数据离散化。通过将属性 A 的值划分为簇或组，聚类算法可以用来离散化数值属性 A。聚类考虑属性 A 的分布以及数据点的邻近性，因此可以产生高质量的离散化结果。遵循自顶向下的划分策略或自底向上的合并策略，聚类可以用来产生属性 A 的概念分层，其中每个簇形成概念分层的一个节点。在前一种策略中，每个初始簇或分区可以进一步分解成若干个子簇，形成较低的概念层；在后一种策略中，通过反复地对邻近簇进行分组，形成较高的概念层。

6. 由标称数据产生概念分层

对于标称数据，概念分层可以基于模式定义以及每个属性的不同值个数产生。使用概念分层变换数据可以发现较高层的知识模式，它允许在多个抽象层进行挖掘。

3.6.2　Python 数据变换与离散化

1. 数据的规范化

数据分析的预处理除了数据清洗、数据合并和标准化之外，还包括数据变换的过程，如类别型数据变换和连续型数据的离散化。

【例 3-37】 数据规范化示例。

```
In[47]:    import pandas as pd
           import numpy as np
           a = [47, 83, 81, 18, 72, 41]
           b = [56, 96, 84, 21, 87, 67]
           data = np.array([a, b]).T
```

```
In[47]:  dfab = pd.DataFrame(data, columns = ['A', 'B'])
         print('最小 - 最大规范化:\n',(dfab - dfab.min())/(dfab.max() - dfab.min()))
         print('零均值规范化: \n',(dfab - dfab.mean())/dfab.std())
Out[47]: 最小 - 最大规范化:
               A         B
         0  0.446154  0.466667
         1  1.000000  1.000000
         2  0.969231  0.840000
         3  0.000000  0.000000
         4  0.830769  0.880000
         5  0.353846  0.613333
         零均值规范化:
                 A           B
         0 - 0.386103  - 0.456223
         1   1.003868    1.003690
         2   0.926648    0.565716
         3 - 1.505803  - 1.733646
         4   0.579155    0.675209
         5 - 0.617765  - 0.054747
```

2. 类别型数据的哑变量处理

类别型数据是数据分析中十分常见的特征变量，但是在进行建模时，Python 不能像 R 语言那样直接处理非数值型的变量，因此，往往需要对这些类别型变量进行一系列转换，如哑变量。

哑变量(Dummy Variables)是用来反映质的属性的一个人工变量，是量化了的自变量，通常取值为 0 或 1。利用 Pandas 库中的 get_dummies()函数对类别型数据进行哑变量处理。

【例 3-38】 数据的哑变量处理。

```
In[48]:  df = pd.DataFrame([
                    ['green', 'M', 10.1, 'class1'],
                    ['red', 'L', 13.5, 'class2'],
                    ['blue', 'XL', 14.3, 'class1']])
         df.columns = ['color', 'size', 'prize','class label']
         print(df)
         pd.get_dummies(df)
Out[48]:    color  size  prize  class label
         0  green     M   10.1       class1
         1    red     L   13.5       class2
         2   blue    XL   15.3       class1
```

输出结果如图 3-14 所示。

对于一个类别型数据，若取值有 m 个，则经过哑变量处理后就变成了 m 个二元互斥特征，每次只有一个激活，使数据变得稀疏。

	prize	color_blue	color_green	color_red	size_L	size_M	size_XL	class label_class1	class label_class2
0	10.1	0	1	0	0	1	0	1	0
1	13.5	0	0	1	1	0	0	0	1
2	15.3	1	0	0	0	0	1	1	0

图 3-14　哑变量处理

3. 连续型变量的离散化

数据分析和统计的预处理阶段，经常会遇到年龄、消费等连续型数值，而很多模型算法（尤其是分类算法）都要求数据是离散的，因此要将数值进行离散化分段统计，提高数据区分度。

常用的离散化方法主要有等宽法、等频法和聚类分析法。

1）等宽法

将数据的值域划分成具有相同宽度的区间，区间个数由数据本身的特点决定或用户指定。Pandas 提供了 cut()函数，可以进行连续型数据的等宽离散化。cut()函数的基础语法格式为

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

cut()函数的主要参数及其说明如表 3-4 所示。

表 3-4　cut ()函数的主要参数及其说明

参　数	说　明
x	接收 array 或 Series，待离散化的数据
bins	接收 int、list、array 和 tuple。若为 int，指离散化后的类别数目；若为序列型，则表示进行切分的区间，每两个数的间隔为一个区间
right	接收 boolean，代表右侧是否为闭区间，默认为 True
labels	接收 list、array，表示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，显示标签的精度，默认为 3

【例 3-39】　cut()函数的应用。

```
In[49]:    np.random.seed(666)
           score_list = np.random.randint(25, 100, size = 10)
           print('原始数据: \n',score_list)
           bins = [0, 59, 70, 80, 100]
           score_cut = pd.cut(score_list, bins)
           print(pd.value_counts(score_cut))
Out[49]:   原始数据:
            [27 70 55 87 95 98 55 61 86 76]
```

```
(80, 100]    4
(0, 59]      3
(59, 70]     2
(70, 80]     1
dtype: int64
```

使用等宽法离散化对数据分布具有较高的要求，若数据分布不均匀，那么各类的数目也会变得不均匀。

2）等频法

cut()函数虽然不能直接实现等频离散化，但可以通过定义将相同数量的记录放进每个区间。

【例 3-40】 等频法离散化连续型数据。

```
In[50]:    def SameRateCut(data,k):
               k = 2
               w = data.quantile(np.arange(0,1 + 1.0/k,1.0/k))
               data = pd.cut(data,w)
               return data
           result = SameRateCut(pd.Series(score_list),3)
           result.value_counts()
Out[50]:   (73.0, 97.0]    5
           (27.0, 73.0]    4
           dtype: int64
```

相比于等宽法，等频法避免了类分布不均匀的问题，但同时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对数据个数的要求。

3）聚类分析法

一维聚类的方法包括两步，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，为合并到一个簇的连续型数据做同一标记。聚类分析的离散化需要用户指定簇的个数，用来决定产生的区间数。

3.7 利用 sklearn 进行数据预处理

sklearn.preprocessing 包提供了一些常用的数据预处理实用函数和转换器类，以将原始特征向量转换为更适合数据挖掘的表示。

sklearn 提供的数据预处理功能如图 3-15 所示。

1. 数据标准化、均值和方差缩放

scale()函数提供一种在单个类似数组的数据集上执行此操作的快速简便方法，其语法格式为

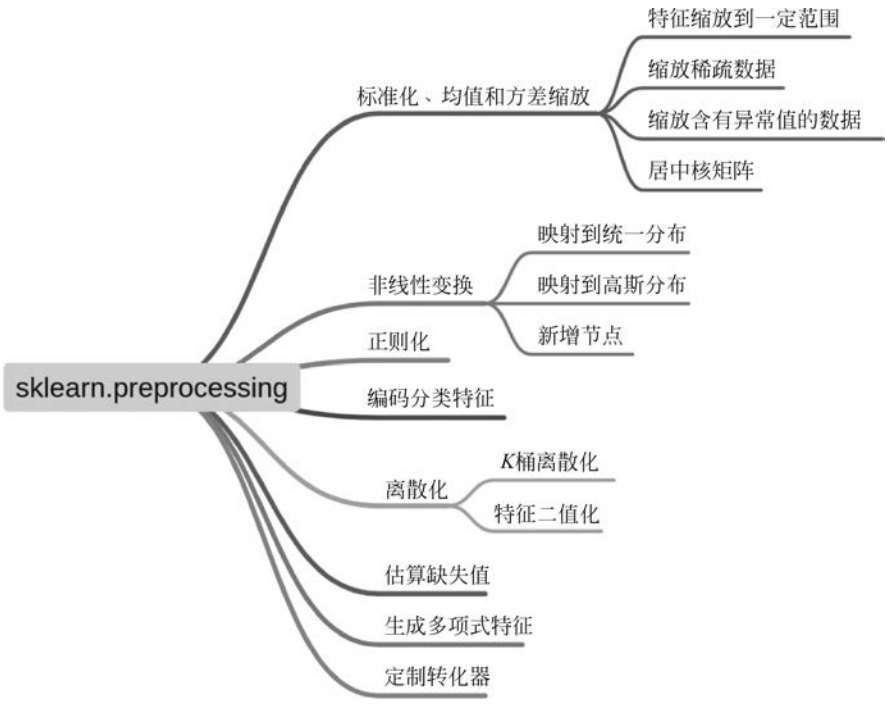

图 3-15　sklearn 提供的数据预处理功能

```
sklearn.preprocessing.scale(X, axis = 0, with_mean = True, with_std = True, copy = True)
```

其主要参数及其说明如表 3-5 所示。

表 3-5　sklearn.preprocessing.scale()函数的主要参数及其说明

参　数	数 据 类 型	说　明
X	{array-like, sparse matrix}	以此数据为中心缩放
axis	int（默认为 0）	轴向设置，0 表示独立地标准化每个特征，1 表示标准化每个样本（即行）
with_mean	boolean（默认为 True）	如果为 True，缩放之前先中心化数据
with_std	boolean（默认为 True）	如果为 True，以单位方差法缩放数据（或者等价地，单位标准差）
copy	boolean, optional,（默认为 True）	如果为 False，原地执行标准化并避免复制

【例 3-41】 数据的标准化、均值和标准差求解示例。

```
In[51]:  from sklearn import preprocessing
         import numpy as np
         X_train = np.array([[ 1., -2.,  1.5],[ 2.2,  1.3, 0.5],[ 0.3,  1., -1.5]])
         X_scaled = preprocessing.scale(X_train)
         print('X_train:\n',X_train)
```

```
            print('X_scaled:\n',X_scaled)
            print('均值: ',X_scaled.mean(axis = 0))
            print('标准差: ',X_scaled.std(axis = 0))
Out[51]:    X_train:
             [[ 1.   -2.    1.5]
             [ 2.2  1.3  0.5]
             [ 0.3  1.   -1.5]]
            X_scaled:
             [[-0.21242964 -1.40942772  1.06904497]
             [  1.31706379  0.80538727  0.26726124]
             [ -1.10463415  0.60404045 -1.33630621]]
            均值: [0. 0. 0.]
            标准差: [1. 1. 1.]
```

preprocessing 模块还提供了一个实用程序类 StandardScaler，用来实现 TransformerAPI 计算训练集的均值和标准差，以便以后能够在测试集上重新应用相同的转换。sklearn 的转换器通常与分类器、回归器或其他评估器组合以构建复合评估器。

【例 3-42】 数据的标准化计算示例。

```
In[52]:    scaler = preprocessing.StandardScaler().fit(X_train)
           print('scaler.scale_ :',scaler.scale_)
           print('scaler.mean_ :',scaler.mean_)
           scaler.transform(X_train)
Out[52]:   scaler.scale_ : [0.78457349 1.48996644 1.24721913]
           scaler.mean_ : [1.16666667 0.1        0.16666667]
           array([[ 1. , -2. ,  1.5],
                  [ 2.2,  1.3,  0.5],
                  [ 0.3,  1. , -1.5]])
```

然后在新数据上使用 Scaler 实例，像在训练集上那样转换它。

```
In[53]:    X_test = [[-1., 1., 0.]]
           scaler.transform(X_test)
Out[53]:   array([[-2.76158538,  0.60404045, -0.13363062]])
```

通过将 with_mean=False 或 with_std=False 传递给 StandardScaler 的构造函数，可以禁用置中或缩放。

2. 特征缩放

另一种标准化是特征缩放，使其介于给定的最小值和最大值之间，通常介于 0 和 1 之间，或者使每个特征的最大绝对值被缩放到单位大小。

1）一般特征值缩放

可以通过 MinMaxScaler 或 MaxAbsScaler 进行一般特征值缩放，具体语句格式为

```
sklearn.preprocessing.minmax_scale(X, feature_range = (0, 1), axis = 0, copy = True)
```

【例 3-43】　数据的缩放示例。

```
In[54]:    X_train = np.array([[ 1., -1.,  2.],[ 2.,  0.,  0.],[ 0.,  1., -1.]])
           min_max_scaler = preprocessing.MinMaxScaler()
           X_train_minmax = min_max_scaler.fit_transform(X_train)
           print('原数据: \n',X_train)
           print('归一化: \n',X_train_minmax)
Out[54]:   原数据:
            [[ 1. -1.  2.]
            [ 2.  0.  0.]
            [ 0.  1. -1.]]
           归一化:
            [[0.5        0.          1.        ]
            [1.         0.5         0.33333333]
            [0.         1.          0.        ]]
```

然后可以在新数据上使用 Scaler 实例，像在训练集上那样转换它。

```
In[55]:    X_test = np.array([[-3., -1.,  4.]])
           X_test_minmax = min_max_scaler.transform(X_test)
           print('测试数据: ',X_test)
           print('归一化的测试数据: \n',X_test_minmax)
           print('',min_max_scaler.scale_)
           print('',min_max_scaler.min_)
Out[55]:   测试数据: [[-3. -1.  4.]]
           归一化的测试数据:
            [[-1.5      0.          1.66666667]]
            [0.5        0.5         0.33333333]
            [0.         0.5         0.33333333]
```

MinMaxScaler 默认转换为[0.0，1.0]，如果 MinMaxScaler 给出一个显式范围 feature_range=(min，max)，则完整的表达式为

```
X_std = (X - X.min(axis = 0))/(X.max(axis = 0) - X.min(axis = 0))
X_scaled = X_std * (max - min) + min
```

MaxAbsScaler 的工作方式类似，但通过除以每个特征中的最大值将训练数据置于[−1,1]。它适用于已经零中心化的数据或稀疏数据。

【例 3-44】　利用 MaxAbsScaler 将数据归一化。

```
In[56]:    X_train = np.array([[ 1., -1.,  2.],[ 2.,  0.,  0.],[ 0.,  1., -1.]])
           max_Abs_scaler = preprocessing.MaxAbsScaler()
           X_train_minmax = max_Abs_scaler.fit_transform(X_train)
           print('原数据: \n',X_train)
           print('归一化: \n',X_train_minmax)
Out[56]:   原数据:
            [[ 1. -1.   2.]
```

```
  [ 2.    0.    0.]
  [ 0.    1.  -1.]]
归一化:
 [[ 0.5  -1.    1. ]
  [ 1.    0.    0. ]
  [ 0.    1.   -0.5]]
```

然后可以在新数据上使用 Scaler 实例，像在训练集上那样转换它。

```
In[57]:   X_test = np.array([[-3., -1.,  4.]])
          X_test_maxAbs = max_Abs_scaler.transform(X_test)
          print('测试数据: ',X_test)
          print('归一化的测试数据: ',X_test_maxAbs)
Out[57]:  测试数据: [[-3. -1.  4.]]
          归一化的测试数据: [[-1.5 -1.   2. ]]
```

2）缩放稀疏数据

将稀疏数据置中会破坏数据的稀疏结构，但是缩放稀疏矩阵又是有意义的，特别是当特征处于不同的缩放比例时。

MaxAbsScaler 和 maxabs_scale 适用于缩放稀疏数据。此外，scale 和 StandardScaler 能够处理 scipy. sparse 矩阵作为输入的情况，此时需要将 with_mean 设置为 False，否则默认的置中操作将破坏数据的稀疏型，会抛出一个 ValueError 错误，而且内存可能会被大量占用造成内存溢出。

需要注意的是，缩放器接受压缩的稀疏行和压缩的稀疏列格式（参见 scipy. sparse. csr_matrix 和 scipy. sparse. csc_matrix）。任何其他稀疏输入都将转换为“压缩稀疏行”表示形式。为避免不必要的内存复制，建议选择上游的 CSR（Compressed Sparse Row）或 CSC（Compressed Sparse Column）表示形式。最后，如果期望置中的数据足够小，则使用稀疏矩阵的 toarray()函数将输入显式转换为数组。

3）带异常值的缩放数据

如果数据中包含许多异常值，那么使用数据的均值和方差进行缩放可能效果不会很好。在这种情况下，可以使用 robust_scale 和 RobustScaler 作为替代，它们对数据的中心和范围使用了更可靠的估计。

3. 非线性变换

非线性变换有分位数变换和幂变换。分位数变换和幂变换都是基于特征的单调变换，从而保持每个特征值的秩。分位数变换将所有特征置于相同的期望分布中。幂变换是一类参数变换，其目的是将数据从任意分布映射到接近高斯分布的位置。

1）映射到均匀分布

QuantileTransformer()函数和 quantile_transform 提供非参数转换，将数据映射到值为 0～1 的均匀分布。

【例 3-45】 将数据映射到值为 0～1 的均匀分布。

```
In[58]:    from sklearn.datasets import load_iris
           from sklearn.model_selection import train_test_split
           X, y = load_iris(return_X_y = True)
           X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 0)
           quantile_transformer = preprocessing.QuantileTransformer(random_state = 0)
           X_train_trans = quantile_transformer.fit_transform(X_train)
           X_test_trans = quantile_transformer.transform(X_test)
           print(np.percentile(X_train[:, 0], [0, 25, 50, 75, 100]) )
           # 此特征对应于以厘米为单位的萼片长度
           print(np.percentile(X_train_trans[:, 0], [0, 25, 50, 75, 100]))
Out[58]:   [4.3 5.1 5.8 6.5 7.9]
           [0.          0.23873874 0.50900901 0.74324324 1.        ]
```

2）映射到高斯分布

在许多建模场景中，数据集中的特性是正常的。幂变换是一类参数的单调变换，其目的是将数据从任意分布映射到尽可能接近高斯分布，以稳定方差和最小化偏度。PowerTransformer 目前提供了两种这样的幂变换：Yeo-Johnson 变换和 Box-Cox 变换。

Box-Cox 变换仅可应用于严格的正数据。在这两种方法中，变换均通过 Lambda 进行参数化，通过最大似然估计来确定。

【例 3-46】 使用 Box-Cox 变换将对数正态分布绘制的样本映射到正态分布。

```
In[59]:    pt = preprocessing.PowerTransformer(method = 'box - cox', standardize = False)
           X_lognormal = np.random.RandomState(616).lognormal(size = (3, 3))
           print(X_lognormal)
           T = pt.fit_transform(X_lognormal)
           print(T)
Out[59]:   [[1.28331718  1.18092228  0.84160269]
            [0.94293279  1.60960836  0.3879099 ]
            [1.35235668  0.21715673  1.09977091]]
           [[  0.49024349   0.17881995 - 0.1563781 ]
            [ - 0.05102892   0.58863195 - 0.57612414]
            [  0.69420009 - 0.84857822   0.10051454]]
```

上述示例中，设置 standardize＝False，PowerTransformer 默认情况下将对变换后的输出应用零均值、单位方差归一化。还可以使用 QuantileTransformer()函数通过设置 output_distribution＝'normal'将数据映射到正态分布。

【例 3-47】 使用 QuantileTransformer()函数进行数据映射。

```
In[60]:    from sklearn.datasets import load_iris
           from sklearn.model_selection import train_test_split
           X, y = load_iris(return_X_y = True)
           quantile_transformer =
           preprocessing.QuantileTransformer(output_distribution = 'normal',
           random_state = 0)
```

```
          X_trans = quantile_transformer.fit_transform(X)
          quantile_transformer.quantiles_
Out[60]:  array([[4.3, 2. , 1. , 0.1],
                 [4.4, 2.2, 1.1, 0.1],
                 [4.4, 2.2, 1.2, 0.1],
                 [4.4, 2.2, 1.2, 0.1],
                 [4.5, 2.3, 1.3, 0.1],
                 [4.6, 2.3, 1.3, 0.2], … ]
```

4. 正则化

正则化的过程是将单个样本缩放到单位范数(每个样本的范数为 1)。如果计划使用点积或任何其他核的二次形式量化任意一对样本的相似性,此过程可能会很有用。该假设是向量空间模型的基础,该向量空间模型经常用于文本分类和聚类。

【例 3-48】 数据正则化示例。

```
In[61]:   X = [[ 1., -1.,  2.],[ 2.,  0.,  0.],[ 0.,  1., -1.]]
          X_normalized = preprocessing.normalize(X, norm = 'l2')
          X_normalized
Out[61]:  array([[ 0.40824829, -0.40824829,  0.81649658],
                 [ 1.        ,  0.        ,  0.        ],
                 [ 0.        ,  0.70710678, -0.70710678]])
```

preprocessing 模块还提供了一个实用程序类 Normalizer,该类使用 Transformer API 实现相同的操作。

5. 编码分类特征

通常,特征不是作为连续的值,而是以绝对的形式给出的。例如,一个人的头发颜色可以是["black", "gray","white"],这些特性可以有效地编码为整数,如取值分别为[0, 1, 2]。若要将分类功能转换为此类整数代码,可以使用 OrdinalEncoder。该估计器将每个范畴特征转换为整数的一个新特征。

【例 3-49】 数据编码示例。

```
In[62]:   enc = preprocessing.OrdinalEncoder()
          X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
          enc.fit(X)
          enc.transform([['female', 'from US', 'uses Safari']])
Out[62]:  array([[0., 1., 1.]])
```

将分类特征转换为可以与 sklearn 估计器一起使用的特征的编码方法称为 One-Hot 编码或 Dummy 编码。可以使用 OneHotEncoder()函数获得这种类型的编码,该编码器将具有 n_categories 个可能值的每个分类特征转换为 n_categories 个二进制特征,其中一个为 1,其他为 0。

【例 3-50】　使用 OneHotEncoder()函数进行分类特征编码示例。

```
In[63]:  enc = preprocessing.OneHotEncoder()
         X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
         enc.fit(X)
         R = enc.transform([['female', 'from US', 'uses Safari'],['male', 'from Europe',
         'uses Safari']]).toarray()
         display(R)
Out[63]: array([[1., 0., 0., 1., 0., 1.],
                [0., 1., 1., 0., 0., 1.]])
```

【例 3-51】　类型数据变换示例。数据集中有两种性别、4 个可能的大洲和 4 个网络浏览器。

```
In[64]:  genders = ['female', 'male']
         locations = ['from Africa', 'from Asia', 'from Europe', 'from US']
         browsers = ['uses Chrome', 'uses Firefox', 'uses IE', 'uses Safari']
         enc = preprocessing.OneHotEncoder(categories = [genders, locations, browsers])
         X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
         enc.fit(X)
         enc.transform([['female', 'from Asia', 'uses Chrome']]).toarray()
Out[64]: array([[1., 0., 0., 1., 0., 0., 1., 0., 0., 0.]])
```

6. 离散化

离散化(也称为量化或绑定)提供了一种将连续特征划分为离散值的方法。某些具有连续特征的数据集可能受益于离散化,因为离散化可以将连续属性的数据集转换为仅具有名义属性的数据集。

One-Hot 编码的离散特征可以使模型更有表现力,同时保持可解释性。例如,用离散化器进行预处理可以将非线性引入线性模型。

1) K 桶离散化

KBinsDiscretizer 将特征离散到 K 个桶(Bin)中。

【例 3-52】　数据的 K 桶离散化示例。

```
In[65]:   X = np.array([[ -3., 5., 15 ],[  0., 6., 14 ],[  6., 3., 11 ]])
          est = preprocessing.KBinsDiscretizer(n_bins = [3, 2, 2],
          encode = 'ordinal').fit(X)
          est.transform(X)
Out[65]:  array([[0., 1., 1.],
                 [1., 1., 1.],
                 [2., 0., 0.]])
```

2) 特征二值化

特征二值化是对数字特征进行阈值化以获得布尔值的过程。

【例 3-53】 特征二值化示例。

```
In[66]:    X = [[ 1., -1.,  2.],[ 2.,  0.,  0.],[ 0.,  1., -1.]]
           binarizer = preprocessing.Binarizer().fit(X)
           Y1 = binarizer.transform(X)
           print(Y1)
           # 可以调整阈值
           binarizer = preprocessing.Binarizer(threshold = 1.1)
           Y2 = binarizer.transform(X)
           print(Y2)
Out[66]:   [[1. 0. 1.]
            [1. 0. 0.]
            [0. 1. 0.]]
           [[0. 0. 1.]
            [1. 0. 0.]
            [0. 0. 0.]]
```

3.8 小结

（1）现实中获得的数据极易受到噪声、缺失值和不一致数据的影响。数据的质量决定了数据挖掘的效果，因此在数据挖掘之前要对数据进行预处理，提高数据质量，从而改善数据挖掘的效果。数据质量可用准确性、完整性、一致性、时效性、可信性和可解释性定义。

（2）数据清洗用于填补缺失值、光滑噪声，同时识别离群点，并纠正数据的不一致性。数据清洗通常是一个两步的迭代过程，即偏差检测和数据变换。

（3）数据集成将来自多个数据源的数据集成为一致的数据存储。实体识别问题、属性的相关性分析是数据集成中的主要问题。

（4）数据标准化用于消除特征之间量纲和取值范围的差异可能会造成的影响，主要包括离差标准化和标准差标准化。

（5）数据归约用于在尽可能保持数据完整性的基础上得到数据的归约表示，主要包括维归约、数量归约和数据压缩等方法。

（6）数据变换是一种将原始数据变换为较好数据格式的方法，以便作为数据处理前特定数据挖掘算法的输入。数据离散化是一种数据变换的形式。

（7）利用 Python 中的 Pandas 和 sklearn 可以方便地实现数据预处理。

习题 3

1. 数据处理中为何要进行数据变换？数据变换的方法主要有哪些？
2. 请分别介绍均值、中位数和截断均值在反映数据中心方面的特点。
3. 现有某班 20 名同学“数据挖掘”课程的成绩，分别为 58，61，67，70，71，75，75，75，

76,77,78,79,79,80,80,81,82,84,88,95,求该组成绩的中位数、众数和极差,并画出该组成绩的箱线图。

4. 数值属性的相似性度量方法有哪些？各自的优缺点是什么？

5. 在数据清洗中,处理数据缺失值的方法有哪些？如何去掉数据中的噪声？

6. 数据离散化的意义是什么？主要有哪些数据离散化方法？

7. 什么是数据规范化？有哪些常用的数据规范化方法？